Event-driven Model Transformations in Domain-specific Modeling Languages

PhD Thesis

István Ráth
MSc in Technical Informatics

Supervisor:
Dr. Dániel Varró, PhD
associate professor

Budapest, March 2011
Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Ráth István Zoltán 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, István Zoltán Ráth, 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, 2011. 03. 31.

Ráth István Zoltán
Acknowledgements

I wish to express my gratitude, first and foremost, to my supervisor, Dr. Dániel Varró. Without his continuous support, valuable insight, ideas, friendly advice and enthusiasm over the past six years, I would have never succeeded.

I am also grateful to my colleagues in the Fault Tolerant Systems Research Group, especially Prof. Dr. András Pataricza and Dr. István Majzik for their trust and support of my research. I highly acknowledge the VIATRA2 team, namely Dr. András Balogh, Gábor Bergmann, Ábel Hegedűs, Ákos Horváth, Zoltán Ujhelyi and Dr. Gergely Varró for the fruitful collaborative research work, and the fun we had. I am thankful to past members András Schmidt and Dávid Vágó for their contributions in the early beginnings. Many thanks to research associates Máté Kovács, Dr. Balázs Polgár, and Dániel Tóth, and my former MSc students: András Ökrös, Ádám Horváth, and all the others for their ideas, suggestions, contributions and hard work in using, testing and developing VIATRA2 over many years.

I would like to thank my colleagues at the University of Leicester, Prof. Dr. Reiko Heckel, Ajab Khan and Dr. Paolo Torrini for the fruitful collaborative work. I also highly acknowledge Philip Mayer at the Ludwig-Maximilians-Universität München for our joint work in the SENSORIA project. I wish to thank Dr. Krzysztof Czarnecki and his group for supporting my research visit at the University of Waterloo.

My work was supported by the SENSORIA (IST-3-016004, FP6), DIANA (AERO1-030985, FP6), SecureChange (ICT-FET-231101, FP7) and MOGENTES (ICT-216679, FP7) European research projects, and the PhD Candidate Scholarship of the Faculty of Electrical Engineering of the Budapest University of Technology and Economics.

Finally, I would like to thank my family and friends for their support and tolerance.
Summary

In today’s model-centric software engineering world, *domain-specific modeling languages* play a pivotal role in agile system modeling, rapid prototyping and generative programming. These techniques are important factors in achieving increased productivity, improved software quality and lower maintenance and documentation costs.

Due to extensive research and industrial interest in language engineering technology over the last decade, the development of domain-specific languages as development tools is nowadays significantly easier, but still requires expertise in a wide range of domains: language engineering heavily relies on core modeling and other auxiliary technologies (such as graphical editing frameworks and parser generators). In parallel, extensive research has been invested into *model transformations* that provide the foundations for integrating DSMLs into development processes and facilitate crucial use-cases such as automated mappings and code generation. The results of these related technologies can be combined to further advance domain-specific language development and decrease tool development costs.

Therefore, this thesis is centered around the concept of tight integration of model transformations into language engineering techniques, so that they may provide high level foundations for advanced language engineering aspects such as well-formedness constraint evaluation, automated mappings between various concrete and abstract syntax representations and simulation based on dynamic semantics.

As a crucial enabling factor, the traditional, batch execution-oriented paradigm of model transformation development needs to be adapted to domain-specific modeling scenarios that are inherently interactive. To support this approach, the core scientific contribution of this work is focused on *event-driven transformations* that run like daemons in the background, and react to complex changes of the model whenever necessary in the form of transformation actions. In this work, I (i) give novel conceptual foundations for event-driven transformations based on event-condition-action formulas that are well known in expert systems; (ii) provide a high level specification language, extending the Viatra2 transformation language to allow a high degree of knowledge and implementation re-use from traditional transformation engineering; (iii) elaborate a high performance prototype implementation of an event-driven graph transformation system based on incremental graph pattern matching.

I developed in-depth applications of this technology in domain-specific modeling language engineering. First, event-driven transformations are combined with specification metamodels to create a mapping library that allows the complete separation of abstract and concrete syntax representations of a modeling language, which is a key challenge in developing very complex, yet still usable DSMLs. I also elaborate a framework for the design-time, integrated execution of dynamic semantics to provide high level model simulation support for DSMLs. By this approach, the rapid prototyping of dynamic modeling languages becomes feasible. Finally, I outline further applications in well-formedness constraint evaluation and incremental code generation.

To illustrate practical applications of the contributions of the thesis, I elaborate tool integration case studies in model-driven software development. In these scenarios, domain integration techniques are extensively used to support systems modeling from various, consistent modeling perspectives, while event-driven transformations are combined with advanced traceability modeling to create change-driven transformations that facilitate change propagation to external (deployed) models.

The results of this work form an integral part of the Viatra2 model transformation framework and the ViatraDSM language engineering framework. The case studies of the thesis are contributions to the SENSORIA, MOGENTES and SecureChange European Union research projects.
Összefoglaló

Napjaink modellközpontú szoftvermérnöki gyakorlatának fontos alkotóelemei a szakterület-specifikus modellezési nyelvek, melyek egy-egy alkalmazási terület fogalmait, az azok között értelmezett kapcsolatokat és az elemek attribútumait írják le. A nyelvekre épülő eszközökkel hatékonyan támogathatók a modern agilis fejlesztési módszertanok, melyek gyors prototipizálás-sal és automatikus kódgenerálással gyorsítják a fejlesztés ütemét. Az elmúlt években történt jelentős fejlesztéseknek köszönhetően ma már a szakterületi modellözönyelvek kifejlesztése sokat egyszerűsödött, de még mindig igen összetett, széleskörű ismereteket igénylő feladat. A gyakorlatban is alkalmazható eszközök készítéséhez elmélyült tudás szükséges mind a metamodellezés, mind pedig a grafikus, illetve szöveges szintaksisú megjelenítést támogató technikák területén (parszer generátorok, grafikus szerkesztők keretrendszer). A nyelvetvezési módszerek fejlődésével párhuzamosan aktiv kutatásmunka folyt a metamodellező és grafikus szintaktikai nyelvek kifejlesztésének területén.

A disszertáció legfontosabb koncepcionális eredménye a hagyományos, kötegelt végrehajtási szemantika helyett javasolt eseményvezérelt transzformációk (event-driven transformations, EDT) technikája, melyek jól illeszkednek a célkitűzésekben felsorolt, alapvetően interaktív felhasználási módon. Az EDT-k az operációs rendszerek háttérben futó szolgáltatásainak megjelenítésére használják közvetlen felhasználói beavatkozás nélkül, folyamatosan futnak, és bármikor képesek a modell tetszőlegesen összetett változásaira reagálni.

A disszertáció fő célkitűzése, hogy e két terület összekapcsolásával olyan módszereket dolgozzon ki, melyek segítségével hatékonyabb és költségtakarékosabb fejlesztéshez készítők a szakterület-specifikus nyelvek támogatására készített modern tervezőeszközök.

A disszertáció legfontosabb koncepcionális eredménye a hagyományos, kötegelt végrehajtási szemantika helyett javasolt eseményvezérelt transzformációk (event-driven transformations, EDT) technikája, melyek jól illeszkednek a céllitűzésekben felsorolt, alapvetően interaktív felhasználási módon. Az EDT-k az operációs rendszerek háttérben futó szolgáltatásainak megjelenítésére használják közvetlen felhasználói beavatkozás nélkül, folyamatosan futnak, és bármikor képesek a modell tetszőlegesen összetett változásaira reagálni.

A disszertáció fő célkitűzése, hogy e két terület összekapcsolásával olyan módszereket dolgozzon ki, melyek segítségével hatékonyabb és költségtakarékosabb fejlesztéshez készítők a szakterület-specifikus nyelvek támogatására készített modern tervezőeszközök.

A disszertáció legfontosabb koncepcionális eredménye a hagyományos, kötegelt végrehajtási szemantika helyett javasolt eseményvezérelt transzformációk (event-driven transformations, EDT) technikája, melyek jól illeszkednek a célkitűzésekben felsorolt, alapvetően interaktív felhasználási módon. Az EDT-k az operációs rendszerek háttérben futó szolgáltatásainak megjelenítésére használják közvetlen felhasználói beavatkozás nélkül, folyamatosan futnak, és bármikor képesek a modell tetszőlegesen összetett változásaira reagálni.

A disszertáció fő célkitűzése, hogy e két terület összekapcsolásával olyan módszereket dolgozzon ki, melyek segítségével hatékonyabb és költségtakarékosabb fejlesztéshez készítők a szakterület-specifikus nyelvek támogatására készített modern tervezőeszközök.

A disszertáció legfontosabb koncepcionális eredménye a hagyományos, kötegelt végrehajtási szemantika helyett javasolt eseményvezérelt transzformációk (event-driven transformations, EDT) technikája, melyek jól illeszkednek a célkitűzésekben felsorolt, alapvetően interaktív felhasználási módon. Az EDT-k az operációs rendszerek háttérben futó szolgáltatásainak megjelenítésére használják közvetlen felhasználói beavatkozás nélkül, folyamatosan futnak, és bármikor képesek a modell tetszőlegesen összetett változásaira reagálni.

A disszertáció fő célkitűzése, hogy e két terület összekapcsolásával olyan módszereket dolgozzon ki, melyek segítségével hatékonyabb és költségtakarékosabb fejlesztéshez készítők a szakterület-specifikus nyelvek támogatására készített modern tervezőeszközök.
## Contents

1 Introduction
   1.1 Model-driven software engineering ................................. 1
   1.2 Domain-specific modeling languages in model-driven software engineering ................................. 4
   1.3 Challenges and contributions ....................................... 6
   1.4 The structure of the thesis ......................................... 10

I Domain-specific language engineering .............................. 13

2 Modeling preliminaries
   2.1 Overview .................................................................. 15
   2.2 Running example ...................................................... 16
   2.3 Models and metamodels ............................................. 17
   2.4 Well-formedness ...................................................... 21
   2.5 Models and metamodels in Viatra2 ............................... 24
   2.6 Traceability models .................................................. 28
   2.7 Summary ............................................................... 29

3 Language engineering in the ViatraDSM framework ............. 31
   3.1 Domains in modeling languages .................................... 32
   3.2 The ViatraDSM Framework ........................................ 35
   3.3 Domain integration techniques ...................................... 39
   3.4 Domain integration over the VPM model space ................ 43
   3.5 Related work ......................................................... 57
   3.6 Summary ............................................................. 59

II Event-driven transformations in domain-specific languages ... 61

4 Preliminaries of model transformations ............................. 63
   4.1 Foundations of model transformations ........................... 63
   4.2 Incremental graph pattern matching .............................. 77
   4.3 Summary ............................................................. 84

5 Event-driven transformations ........................................... 85
   5.1 Foundations of incremental model transformations .......... 86
   5.2 Overview of event-driven transformations ....................... 89
   5.3 Graph triggers: an event-driven transformation language .... 98
CONTENTS

5.4 Performance evaluation .......................................................... 102
5.5 Related work ........................................................................ 107
5.6 Summary .............................................................................. 109

6 Synchronization between abstract and concrete syntax ............. 111
6.1 Introduction .......................................................................... 112
6.2 Challenges of model synchronization in graphical editors ......... 113
6.3 Modeling the Abstract and Concrete Syntax ......................... 116
6.4 Generic abstract-concrete synchronization with mapping models and live transformations ........................................... 117
6.5 Arbitrary abstract–concrete syntax mapping .......................... 122
6.6 Implementation details ............................................................ 126
6.7 Related work ........................................................................ 128
6.8 Summary .............................................................................. 129

7 Model simulation in DSMLs ..................................................... 133
7.1 Overview .............................................................................. 134
7.2 Model simulation by event-driven transformations .................... 134
7.3 Performance evaluation ............................................................. 139
7.4 Related work ........................................................................ 142
7.5 Summary .............................................................................. 143

III Applications in tool integration ........................................... 145

8 Change-driven transformations .............................................. 147
8.1 Overview .............................................................................. 148
8.2 Change history models .............................................................. 150
8.3 Applying CHMs to non-materialized models ......................... 157
8.4 Related work ........................................................................ 158
8.5 Summary .............................................................................. 161

9 Tool integration based on change-driven transformations ......... 163
9.1 The architecture of tool integration chains .............................. 164
9.2 A metamodel for development and integration processes ........ 166
9.3 Case Study: The DIANA toolchain ....................................... 168
9.4 Implementation of the Tool Integration Framework ................ 172
9.5 Process modeling languages for tool integration .................... 174
9.6 Related work ........................................................................ 176
9.7 Summary .............................................................................. 178

IV Conclusions and Appendix .................................................. 179

10 Conclusion .............................................................................. 181
10.1 Fulfillment of objectives ........................................................... 181
10.2 Applications of new results ...................................................... 182
10.3 Future research directions ....................................................... 183
## CONTENTS

**A  Model transformation source listings**  
A.1 Abstract-concrete syntax synchronization transformations  

**B  Stochastic model simulation**  
B.1 Introduction  
B.2 Stochastic execution of simulation schemes  
B.3 Prototype implementation  
B.4 Initial experiments with simulating P2P networks  

**C  Tool integration in the SDE**  
C.1 Introduction  
C.2 High-Level Overview  
C.3 Design and Implementation  
C.4 Integrated Tools  
C.5 Tool Applications  
C.6 Summary  

List of publications  

Bibliography  

---  

A  Model transformation source listings  
185  
A.1 Abstract-concrete syntax synchronization transformations  185  

B  Stochastic model simulation  
189  
B.1 Introduction  189  
B.2 Stochastic execution of simulation schemes  191  
B.3 Prototype implementation  192  
B.4 Initial experiments with simulating P2P networks  193  

C  Tool integration in the SDE  
195  
C.1 Introduction  195  
C.2 High-Level Overview  196  
C.3 Design and Implementation  198  
C.4 Integrated Tools  203  
C.5 Tool Applications  206  
C.6 Summary  210  

List of publications  
211  

Bibliography  
215
Chapter 1

Introduction

1.1 Model-driven software engineering

Models are projections of reality. In engineering, they are used to capture concepts related to the engineer’s task: the product that is being designed, its environment, and processes that may be involved in its life cycle. Models are attractive because they allow a problem to be precisely grasped at the right level of abstraction without unnecessarily delving into detail.

In software development, models are used to represent data structures, communication between agents, algorithms, or at a higher abstraction level, entities in a complex system, e.g. organisational units or products in an e-commerce application. The software industry has been experimenting with high level models at the core of development for two decades now, pursuing the promise that software systems of large complexity can be designed and maintained at a significantly lower cost, if the level of abstraction for engineering design is set considerably higher than that of conventional programming languages.

As models (in their abstract form) are directly meaningful only to humans, they need to be translated into the language that the target platform (software and hardware infrastructure) can understand and run. For low level models, this translation is usually called compilation (in this context, a programming language construct, i.e. source code is also considered a model); while for high level models, the term model transformation is frequently used. This is traditionally called model-to-model transformation. In contrast, a special case of model transformation is referred to as code generation, where source code is generated from a (graphical) model, using a code generator (model-to-text transformation).

The model-driven software development (MDSD) paradigm is based on the idea that the developer should work with high abstraction level models during most of the development cycle, and the rest of the work should be automated to the highest possible degree by advanced tools. Source code, the traditional end product of software development, should be generated to the largest possible extent, to minimize the amount of business logic that is outside of the scope of modeling, and is only represented by handwritten code. Additionally, the abstract models should also be used to enhance the overall quality of the product, by supporting early design analysis using formal and semi-formal techniques [VP03] such as model checking, static analysis [BDG+06, KJG05] and model-based test generation [MPB02].
1.1.1 Steps of evolution

Early beginnings: CASE tools

Computer aided software engineering (CASE) tools first emerged during the 1980’s, when many software development companies realized that meeting increased demand for high quality and complex software required more sophisticated development methods than those used previously.

The main problem with these early tools was the lack of a common approach. Although the methodologies they were built to support shared similar concepts, on the implementation level they differed substantially. Due to the lack of a common graphical notation system, even development documentation was hardly reusable, which directly contributed towards their failure.

Standardization: UML

The Unified Modeling Language (UML) was conceived to provide a common framework for specification, modeling, and documentation in the software development process. In many senses, it was a success (especially when compared to CASE), because it established a standardized, visual and easy-to-use notation system which was comprehensive enough to capture all major aspects of software engineering. Today, UML is the industry standard for software modeling and specification.

As UML by itself is only a documentation and modeling standard, early UML tools were mere auxiliary utilities rather than a central key technology of model-driven development. Still – especially with its major redesign with version 2.0 – it provided good foundations for further development, as today’s advanced tools (such as IBM Rational Software Architect) feature some support for code generation and reverse engineering.

A process design pattern: MDA

Motivated to progress beyond UML’s focus on documentation, the Object Management Group (OMG) issued the Model Driven Architecture (MDA) standard. MDA is essentially an approach to model-based software development utilizing OMG’s flagship technologies, UML, the Meta Object Facility (MOF), XML Metadata Interchange (XMI), and the Common Warehouse Metamodel (CWM). MDA provides a template for model-driven development processes and summarizes best practices and design patterns.

As shown in its overall architecture (Figure 1.1), MDA emphasizes the clear distinction between Platform Independent Models (PIM) and Platform Specific Models (PSM), thus, software development in MDA is envisioned as a three-step process. First, the Platform Independent Model is designed, which is supposed to use modeling concepts which are not platform specific. The second step is to generate a Platform Specific Model (PSM), which contains additional UML models, and represents an implementation of the system under design which can run on the target platform. The transitions between PIM and PSM are facilitated using automated model transformation technology. Finally, application code is generated from the Platform Specific Model.

Model transformations in MDA

Model transformation plays a key role in the overall process of MDA. The aim of model transformation is to carry out automated translations within and between modeling languages. By model transformations, information can be propagated within models on the same level of abstraction (e.g. model synchronization between PIM aspects), moreover, automated transformation tools also provide support for writing translations that cross abstraction level boundaries (e.g. PIM-to-PSM mappings).
These mappings are used in a wide spectrum of applications, e.g. for code generation, to map system models into mathematical domains for early analysis, various model management tasks such as version migration, support for model-based test generation, model composition and well-formedness checking, and reverse engineering of source code into high level models [BBFG08].

Ideally, MDA tools should provide support for creating and editing models, checking completeness and consistency, calculating metrics, transforming models to other models or program code, composing several source models, model-based testing, simulating the execution of the systems represented by models, and reverse engineering by transforming legacy systems to well-formed models. In practice, however, tools are rather fragmented, and while some excel in one or two of the above features, there is still no universally accepted platform that covers the entire feature spectrum.

1.1.2 Problems with UML and MDA

The practical problems of state-of-the-art (UML-centric) MDA tools can be traced back to three major factors:

- **Inflexible customization**
  MDA promotes that domain knowledge should be captured as high level UML PIMs. In practice, however, UML’s facilities (stereotyping and profiles) fell short as it is very hard (expensive) to create custom sub-languages or dialects that are tailored to the needs of individual organizations or their concrete projects. UML, by its nature, was intended to be a general purpose software modeling language, and as such, is not simple enough to be efficiently used by non-software experts (which would be a key success factor for agile development).

- **Imprecise semantics**
  Furthermore, as revealed by a number of surveys [Kob99], even the most recent versions of the
UML standard suffer from multiple ambiguities and problems regarding the precise semantics of the language. This is a major problem as model-centric development highly relies on precise modeling techniques since the information contained in models will be directly propagated to application code at the end of the tool chain. While this is a widely known problem (which has spawned a number of initiatives to fork the UML into more precisely specified variants [Theh]), still none of these gained enough inertia to motivate the standardizing body to significantly improve the UML in this respect.

- **Incomplete support for model transformations**

  The multi-level modeling-based architecture of MDA, as emphasized before, highly relies on transformations within and between different models and languages. Even though the OMG issued the QVT Standard [The08a] as a high-level specification for a model transformation language family, it did not yet gain wide adoption: industrial grade QVT tools are still in early stages of development, as the model transformation community was hesitant to adapt its technologies due to problems and ambiguities of the standard. Instead, it produced a number of tools that are characterized by a wide spectrum of theoretical bases and engineering background technologies. Unfortunately, the scalability and industrial applicability of these technologies varies greatly.

**Pragmatics: Domain-Specific Modeling**

Despite the debate about the strengths and weaknesses of UML and MDA, industrial practice has proven that modeling can bring significant business advantages to those who master it [FC04, Com03]. One of the leading trends in model-centric development today is the usage of *domain-specific modeling languages (DSMLs)* that are analogous to domain-specific languages (such as SQL) in the sense that in contrast to a general-purpose language, they are suited to express the notions of a focused problem domain much more precisely (on the expense that they are not well suited for usage in other domains). The origins of domain-specific languages can be traced back to the specialized programming languages of the 1960’s and 70’s, and the more recent initiatives of intentional programming [Sim95] and aspect-oriented programming [KLM+97], but the key emphasis in today’s DSM technology is that modern tools are directed towards domain experts rather than programmers.

Thus, the main advantage of DSMLs over UML is the ability to capture domain knowledge concisely, at the right level of abstraction, so that domain experts can be directly involved in the development process. Moreover, domain-specific models have precise semantics (as their focus on the problem domain is tighter, such semantics are usually easier to specify than for general-purpose languages), and are thus well suited for automated processing and code generation. DSM is a top-down and vertical approach: instead of trying to create high abstraction level “interfaces” to source code, it gives the designer the freedom to use structures and logic that is specific to the target application domain, and thus, completely independent of programming language concepts and syntax. Early design analysis can also be more focused by exploiting domain-specific abstractions.

### 1.2 Domain-specific modeling languages in model-driven software engineering

In practical systems engineering, DSM concepts are frequently combined with other practices such as the MDA. A typical adaptation pattern is shown in Figure 1.2.
1.2. DOMAIN-SPECIFIC MODELING LANGUAGES IN MODEL-DRIVEN SOFTWARE ENGINEERING

Here, domain engineers capture the core constructs of the system under design by using visual design environments offering rich visual editors tailored to the specific application domain. These visual (and textual) languages specify various (inter-dependent) design aspects of the platform-independent model (PIM) and they may rely on standardized, general-purpose languages such as UML and UML profiles, and also on custom-made DSMLs specialized for the needs of the organization or the concrete system-under design itself.

The information contained in this complex PIM is then further processed along the development chain. Instead of a single family of PSMs, various intermediate models may be used (which are frequently DSMs) that represent various stages and aspects of development (such as test models, deployment configurations, architecture prototype models, low-level models of source code to be generated etc). Finally, deployment artefacts and source code is generated from these low-level models.

**Language engineering in development processes** While DSMLs provide great flexibility for rapid software development, the main challenge of their application is the relative difficulty and expensive-ness of language engineering that becomes part of the software development process. In other words, modern development tools are increasingly used not only for developing the target application itself, but also the auxiliary tools, plug-ins and generators that aid agile development and later re-use.

Even though nowadays there are plenty of ready-made DSMLs available (some of which have been recognized as international industry standards, such as SysML, AADL, AUTOSAR, and UML-MARTE in embedded software development, or BPEL, BPMN2 and SoaML in service-oriented business systems engineering, to name a few well-known examples), in many cases they are not a perfect fit for a particular software development problem. In such cases, DSMLs have to be developed or customized using language engineering tools such as the Eclipse Modeling Framework (EMF) [?].
Such language engineering frameworks are used to create prototype modeling environments rapidly, and then to develop prototypes into fully featured development tools that can be integrated into complete toolchains. These tools, often integrated into development environments (such as Eclipse, in the case of EMF and its counterpart, the Graphical Modeling Framework [GMF]), have various degrees of support to traditional aspects of language engineering such as:

- **Concrete syntax**, by which the human interface of the language (graphics or textual notation) are specified. For instance, a textual language engineering tool can be used to specify the grammar the symbols of the language, or graphical primitives may be assigned to notions and concepts.

- **Abstract syntax** that maps the symbols of the language to the underlying (metamodelling) framework. Abstract syntax representations are usually not user-friendly (in order to ease automated model processing) and have to follow stricter rules. Metamodelling is nowadays synonymous to abstract syntax specification as all major language engineering tools rely on this approach.

- **Mappings between representations** specify how abstract and concrete syntax models are interconnected, e.g. how a visual diagram rendering is (automatically) mapped to an instance model, or how strings are parsed (or how models are rendered into strings). The flexibility of the mapping technology has an elementary impact on the abstraction capabilities of the DSML, as it determines e.g. how much of the information contained in the abstract model can/has to be represented in the human-readable form.

- **Well-formedness rules**, which augment the abstract syntax to express more complicated, domain-specific constraints on the models. For instance, a constraint specification language such as OCL is frequently used for this purpose, and the language engineer uses that language to describe erroneous model configurations (in terms of the metamodel). These constraints will then be evaluated on instance models.

- **Dynamic semantics (for simulation)** that specify behavioral aspects of DSML. Such aspects may be expressed in terms of a well-known dynamic language such as statecharts (denotational semantics), or using a lower-level model manipulation language (operational semantics). Advanced environments, which support the in-design execution of such rules, enable designers to visualize the system-under-design as it progresses through its dynamic states. For instance, a state chart simulator allows the designer to initiate execution runs, step through consecutive states, visualize the execution trajectory using state variable value traces etc.).

### 1.3 Challenges and contributions

Unfortunately, despite the significant investment of research and development effort into state-of-the-art industrial language engineering tools (e.g. EMF has been around since 2003, with numerous major releases), language engineering remains an expensive, time-consuming and many times cumbersome development task that requires special skills [KRA+10].

Interestingly – and analogously to the situation with MDA –, one of the key areas, where *industrial* DSM technology also falls short, is the lack of easy-to-use model transformations. As a result, DSML-based development tools created with language engineering frameworks are *static and closed* in the sense that:
While most tools have advanced support for writing code generators, it is non-trivial to integrate models into a library of other modeling languages \cite{ZKD+10} – in many cases, ad-hoc technologies (operating on low-level XML representations) are used for this purpose. Such solutions are very hard to re-use.

In many cases, they have limited abstraction capabilities between abstract and concrete representations (this is especially true of wide-spread graphical DSMLs based on the GMF); which limits the usability of graphical languages.

Language engineers have to learn complicated languages such as OCL \cite{MDT} in order to specify even simple well-formedness constraints that cannot be expressed in the abstract syntax specification language (metamodel). This frequently leads to the omission of such constraints from end products, and has a serious impact on the quality of the tool.

Many DSMLs are restricted to static system modeling, as it is challenging to create dynamic languages that incorporate behavioral aspects since the language prototyping tool rarely provides easy-to-use means to animate and/or simulate the models directly, within the modeling environment. While there are excellent dedicated tools for dynamic languages (e.g. MATLAB), in many cases they prove to be too expensive or complicated to use for a simple DSML prototype.

In current DSML tools, language engineering aspects related to model processing are covered by heterogeneous and isolated formalisms and mechanisms, which vary greatly from tool to tool. Thus, customizable features or extensions depend either on the public programming interface or on the export-import facilities, and there is no end-to-end approach that covers integration aspects such as mappings between languages, model simulation support for dynamic analysis, or efficient evaluation of complex constraints. These are all hindering factors in DSML adaptation, since they considerably raise development costs.

In this thesis, I argue that model transformation technology can be adapted to the specific needs of domain-specific modeling language engineering, in order to provide support for advanced design aspects, which may significantly improve the capabilities of custom-built tools and reduce the required effort at the same time.

1.3.1 Challenges

The problem areas identified previously are formulated as research challenges as follows:

- **Challenge 1: Lack of a uniform approach to multi-domain modeling.** Domain-specific language engineering frameworks are focused on the development of a single language, or a family comprised of loosely connected languages. Mappings between identical or similar concepts (that may be present in multiple domains at the same time) are difficult to specify and maintain consistently. As most modeling environments are statically typed, model objects that represent information relevant in multiple domains (system model views) have to be replicated, which raises consistency and redundancy issues.

- **Challenge 2: Limitations of abstract and concrete syntax representation in current domain-specific modeling language engineering frameworks.** State-of-the-art visual modeling language engineering technology imposes limitations on how abstract and concrete syntax models are represented. In simplistic cases (e.g. MetaEdit+ \cite{MEP}), these two model representations
are connected by direct one-to-one links. In more advanced such as the GMF, abstract and concrete syntax models are stored in two distinct modeling layers, however there is a complicated synchronization mechanism between the two, limiting the ability of the designer to (i) create an arbitrary concrete syntax representation for a (pre-existing) language, or (ii) design a more complex abstract syntax for a given visual language. In practice, this reduces the scalability and usability of domain-specific modeling languages, especially as DSMLs are growing larger and more complex.

- **Challenge 3:** Lack of integrated generic support for the specification and execution of design-time, interactive simulation, parameterized by the behavioral semantics of the language. Traditional DSM language engineering frameworks lack support for the specification and design-time execution of dynamic semantics. Even though many model transformation tools may be used as alternatives or complementary technologies to dedicated model simulation tools, they are not intrinsically targeted at language engineering, so they are typically used as complementaries to DSM frameworks. This raises integration issues which is ultimately a prohibiting factor in the rapid prototyping of dynamic languages.

- **Challenge 4:** Lack of techniques to specify reactions to complex model changes in incremental transformations. As DSMLs are interactive tools where the models are constantly changing, the adaptation of automated model transformation techniques to support advanced DSMLs are currently impaired by a lack of support for the transparent detection and processing of model changes. While incremental model transformations (e.g. model synchronization) can be implemented using state-of-the-art tools, these only support the processing of elementary model changes (or heavily rely on extensive traceability models to calculate changes that are to be processed). Furthermore, the detection of changes relies on customized (user interface, or tool-specific) functionality, so generic change processing transformations are not feasible.

- **Challenge 5:** No built-in support for incremental model synchronization and incremental code generation. Even if complemented with state-of-the-art model transformations as auxiliary technology, language engineering environments lack facilities for the seamless integration of transformations to facilitate advanced features such as incremental, on-the-fly model synchronization or incremental code generation.

### 1.3.2 Contributions

To answer the research challenges, I propose advanced foundational techniques for the integration of model transformations to language engineering practice. I apply these techniques and develop core technologies that support (i) the complete separation of abstract and concrete syntax representations by incremental model synchronization; (ii) interactive, design-time and scalable simulation of dynamic semantics; (iii) incremental code generation. To support such integrated and interactive application of transformations, I develop a novel, event-driven execution scheme. I also examine the practical considerations of these techniques with special regard to systematic performance evaluation by benchmarking. Finally, I describe the application of the core research results in practical tool integration. I demonstrate all the concepts and contributions on a basic language engineering example centered on the Petri net domain [JKW02].

My contributions are the following:
Foundations of model transformations

- **Contribution 1**: Event-driven execution for incremental model transformations. Model transformations have traditionally been implemented as batch programs using explicit (e.g. scripted) or implicit (e.g. "as-long-as-possible") control flow. Unfortunately, this approach does not align well with interactive application scenarios where the system is in constant interaction with the user (or another system) and it has to provide immediate reactions to events. To overcome this, I elaborate the concepts for event-driven graph transformations based on incremental pattern matching, where changes of the model graph are detected by changing graph pattern match sets that represent compound changes of arbitrary complexity. Event-driven transformations are specifically targeted towards interactive modeling scenarios, where transformations are performed on-the-fly in response to some modifications or changes of the underlying model (independently of the originator of the change). Such scenarios include model-to-model synchronization, consistency and traceability management between two views of a system model, live well-formedness constraint evaluation (e.g. if the violation feedback needs to be shown to the user as the changes happen), or the execution of dynamic semantics directly within an interactive design-time simulator (e.g. if the system needs to update enabled simulation rules as the model is evolving).

- **Contribution 2**: Change-driven model transformations in tool integration applications. I propose the novel concept of change-driven transformations (CDTs), which operate on changes (represented as serialized change models or event objects) and decoupled host models, in order to allow asynchronous change propagation between non-materialized models. I apply this technology to non-intrusive incremental code generation, in the context of a novel model-based tool integration framework, where transformations can be used transparently as co-operating services. Information transfer between automated and semi-automated activities is facilitated using CDTs.

DSM-specific contributions

- **Contribution 3**: The integration of multi-aspect domain-specific language engineering with model transformations. I propose a metamodeling framework, based on the foundations of Viatra2 and ViatraDSM, to support multi-aspect modeling for domain-specific languages through a uniform modelspace, where both the meta- and instance models for multiple domain-specific modeling languages can be persisted, allowing for viewing the same instance models from different domain-specific perspectives (multi-domain integration). This rapid prototyping environment for language engineering is supported by the model transformations of the Viatra2 framework, and thus allows for a transformation-based approach to advanced language engineering aspects.

- **Contribution 4**: Complete separation of abstract and concrete syntax representation in DSMLs, by event-driven model transformations. I elaborate a technique for the complete separation of abstract and concrete syntax of domain-specific modeling languages based on generic event-driven model transformations, which allow for significantly enhanced abstraction capabilities for designing graphical DSMLs with increased usability.

- **Contribution 5**: Design-time simulation of dynamic DSMLs based on event-driven transformations. I propose a novel approach for the design-time simulation of visual domain-specific
models based on event-driven simulation rules captured by formal specification techniques, to provide high-level support for debugging the dynamic semantics of executable domain-specific languages within the editing environment itself.

1.4 The structure of the thesis

The thesis is structured into four parts containing ten chapters (including this introduction) that contain overviews and new results, and three appendices complementing the main parts with additional information.

- **Part 1: Domain-specific language engineering**
  - Chapter 2 establishes metamodeling foundations used throughout the thesis.
  - Chapter 3 introduces the ViatraDSM framework and details the contributions of this work in multi-domain integration (Contribution 3 answering Challenge 1).

- **Part 2: Event-driven transformations in domain-specific languages**
  - Chapter 4 describes preliminaries and concepts related to model transformations, which will be used as foundations and examples later on.
  - Chapter 5 proposes a novel, event-driven specification and execution scheme for model transformations (Contribution 1 providing solutions to Challenge 4).
  - Chapter 6 presents the applications of the event-driven transformation technology to abstract-concrete syntax synchronization in DSMLs (Contribution 4 answering Challenge 2).
  - Chapter 7 presents the adaptation of event-driven transformations to design-time execution and simulation of dynamic semantics (Contribution 5 to solve Challenge 3).

- **Part 3: Applications in tool integration**
  - Chapter 8 describes the generalization and extension of event-driven transformations to change-driven transformations using special traceability models. This chapter also discusses the applications of this technology to incremental code generation (Contribution 2 to Challenge 5).
  - Chapter 9 presents a novel tool integration framework that leverages the results of the thesis in both domain-specific modeling and event-driven model transformations.

- **Part 4: Conclusions and Appendix**
  - Chapter 10 concludes the main parts of the thesis and gives an overview of applications of the results and outlines future directions for research.
  - Appendix A contains listings of some of the longer transformation source code examples.
  - Appendix B describes an application example of the dynamic semantics simulation technology for stochastic system simulation, developed in joint work with the University of Leicester.
  - Appendix C describes the SENSORIA Development Environment, a tool integration framework, developed in joint work with Philip Mayer’s team at the Ludwig-Maximilians-Universität München. The results presented in Chapter 9 are based on this technology.
Notational guide

In order to maintain a consistent textual style of the thesis, the following rules are followed:

- This thesis is mainly written in third person singular. In conclusions after each chapter, I emphasize my own contribution by first person singular or plural.
- Terms in formal definitions are printed in **bold** letters.
- Code extracts always appear as *typewritten* text in listings with grey background.
- References to (modeling) concepts are typeset in *italics*; types in metamodels (esp. when referring to text in figures) are typeset using sans fonts.
- References to own publications appear as bold numbers (e.g. [3]), while citations from the bibliography are formatted alphanumerically (e.g. [VP03]).
Part I

Domain-specific language engineering
Chapter 2

Modeling preliminaries

2.1 Overview

Recently, the main trends in model-driven software engineering have been dominated by the Model Driven Architecture (MDA) [MDA01] vision of the Object Management Group (OMG). According to MDA, model-driven software development begins with a thorough modeling phase where first (i) a platform independent model (PIM) of the business logic is constructed from which (ii) platform specific models (PSMs) including details of the underlying software architecture are derived by model transformations followed by (iii) an automatic generation of the target application code.

The PIMs and PSMs are defined by means of the Unified Modeling Language (UML) [Obj03b], which has become the de facto standard visual object-oriented modeling language in systems engineering with a wide range of applications. Its major success is originating in the fact that UML (i) is a standard (uniformly understood by different teams of developers) and visual language (also meaningful to customers in addition to system engineers and programmers).

However, based upon academic and industrial experiences, several surveys of the past years (such as [Kob99]) have pinpointed several shortcomings of the language concerning, especially, its imprecise semantics, and the lack of flexibility in domain specific applications. In principle, due to its in-width nature, UML would supply the user with every construct needed for modeling software applications. However, this leads to a complex and hard-to-implement UML language, and since everything cannot be included in UML in practice, it also leads to local standards (profiles) for certain domains.

Hence, as complementary technologies to UML, metamodeling language families such as the ECore Language of the Eclipse Modeling Framework [?] have been developed. These developments were primarily motivated by the needs for better support for domain-specific modeling languages (DSMLs) (see Chapter 3 for an in-depth discussion), i.e. to give more flexibility and control over how models can be constructed and processed, to language engineers in charge of creating and adapting custom modeling languages to concrete software development processes.

Metamodeling frameworks that are derived (with slight variations) from the Meta Object Facility (MOF) [Obj03a] metamodeling standard (e.g. ECore) are modular in nature: they have built-in facilities to host custom modeling languages on top of a kernel language. However, as stated in [VP03], MOF fails to support multi-level metamodeling, which, in many practical applications, leads to a number of critical shortcomings that are hard to overcome (e.g. a typical problem area is the integration of different technological spaces where different metamodeling paradigms – e.g. EMF and XML
Schemas – are used in conjunction).

Therefore, the VPM (Visual and Precise Metamodeling) [VP03] metamodeling approach was chosen in the Viatra2 framework, which can support different metamodeling paradigms by supporting multi-level metamodeling with explicit and generalized instanceOf relations. In this work, we rely on a slightly modified (simplified) version of the original VPM approach, that was formally described in [VB07].

2.2 Running example

Throughout the thesis, we use Petri nets (formally, Place/Transition nets with inhibitor arcs) as a demonstrating example.

Example 1 (Petri nets) Petri nets are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. From a system modelling point of view, a Petri net model is frequently used for correctness, dependability and performance analysis in early stages of design.

Petri nets are bipartite graphs, with two disjoint sets of nodes: Places and Transitions. Places may contain an arbitrary number of Tokens. Tokens are also modeled as objects to support visual representation. The state of the net can be changed by firing enabled transitions. A transition is enabled if each of its input places contains at least one token and no place connected with an inhibitor arc contains a token (if no arc weights are considered). When firing a transition, we remove a token from all input places (connected to the transition by Input Arcs) and add a token to all output places (as defined by Output Arcs).

![Figure 2.1: A sample Petri net](image)

Definition 1 (Directed and Labeled Graph) A directed and labeled graph (denoted by $G = (V_G, E_G, src_G, trg_G, lbl_G)$) is a 5-tuple, where $V_G$ and $E_G$ represent the set of nodes and edges of the graph, respectively. Functions $src_G : E_G \rightarrow V_G$ and $trg_G : E_G \rightarrow V_G$ map edges to their source and target node, respectively. Function $lbl_G : V_G \rightarrow$ String maps graph nodes to their label (encoded as a String of characters).

Definition 2 (Bipartite graph) A bipartite graph is a directed graph with two disjunct sets (partitions) of nodes $V_G^1, V_G^2 \subseteq V_G$, where edges may only connect nodes belonging to different partitions. Formally, $\forall e \in E_G : (src_G(e) \in V_G^1 \land trg_G(e) \in V_G^2) \lor (src_G(e) \in V_G^2 \land trg_G(e) \in V_G^1)$
2.3 MODELS AND METAMODELS

Definition 3 (Petri net) A simple Petri net $PN$ is a bipartite labeled graph with disjunct node sets $P$ (places) and $T$ (transitions): $P \cup T = V_G$. The edges are grouped into disjunct sets $IA$ (input arcs) and $OA$ (output arcs): $IA \cup OA = E_G$, and input arcs may be inhibitor arcs: $InhA \subset IA$. Input and inhibitor arcs are leading from places to transitions: $\forall e \in IA : src_{G}(e) \in P \land trg_{G}(e) \in T$, and output arcs are leading from transitions to places: $\forall e \in OA : src_{G}(e) \in T \land trg_{G}(e) \in P$. Additionally, each place contains an arbitrary (non-negative) number of tokens, represented by string labels.

Example 2 (Petri net dynamic semantics) A distribution of tokens over the places of a net is called a marking. The dynamic semantics of the language is defined as a relation on its markings, i.e. the changes in the distribution of tokens over time, starting from the initial marking $M_0$.

A transition of a Petri net may fire whenever there is a token at the start of all input arcs; when it fires, it consumes these tokens, and places tokens at the end of all output arcs. A firing is atomic, i.e., a single non-interruptible step. The execution of Petri nets is nondeterministic: when multiple transitions are enabled at the same time, any one of them may fire. If a transition is enabled, it may fire, but it doesn’t have to.

Since firing is nondeterministic, and multiple tokens may be present anywhere in the net (even in the same place), Petri nets are well suited for modeling the concurrent behavior of distributed systems.

2.3 Models and metamodels

In this section, we conceptually follow [VB07] in defining the abstract syntax of the VPM modeling language. A precise formalization based on a minimal subset of MOF concepts and basic mathematical notions (i.e. sets, relations, functions and tuples) is established in [VP03]. In this work, we present an evolution of the original VPM formalization that corresponds to the current modeling infrastructure of the ViATRA2 framework.

Standard metamodeling paradigms can be integrated into ViATRA2 by import plugins and exporters defined by code generation transformations. So far, models from very different technological spaces such as XML, EMF, semantic web and modeling languages of high practical relevance like BPEL [HBRV10], UML (and various domain-specific languages in the dependable embedded, telecommunication, and service-oriented domain) have been successfully integrated into ViATRA2. While ViATRA2 offers the VTML language for constructing models and metamodels, the main usage scenario is to bridge heterogeneous off-the-shelf tools by flexible model imports and exports.

2.3.1 Model store

The VPM language handles models as the basic units of information storage in a model store. Model stores are universal containers, which can contain any number of models. A model in a model store can be a single atom as well as a complex structure of elements. The VPM model store supports the declaration of supertyping and instantiation relationships, that correspond to the common object-oriented programming notions of inheritance and class-object instantiation.

Definition 4 (Model store) A model store $MST = (M, instanceOf, supertypeOf)$ is a triple, defined by a set of models $M$, and predicates $instanceOf : M \times M \rightarrow \text{Boolean}$ and $supertypeOf : M \times M \rightarrow \text{Boolean}$ that represent instantiation and supertyping relationships, respectively.
In the VPM syntax, metamodels (classes) and instance models (objects) are handled uniformly in a flexible arrangement, which means that there is no fundamental difference between metalevels, any model can play the role of a metaclass and an instance (simultaneously). Being handled uniformly, all models are expressed in terms of a minimal kernel modeling language (called the core metametamodel) that enumerates the basic notions and relationships of the model store.

Definition 5 (Core Metametamodel) The unique core metametamodel \( CMM \) is a special model in a model store \( MST \) that is at the top of the instantiation and the supertyping hierarchy. Formally, \( \forall M : \text{instanceOf}(CMM, M) \lor \text{supertypeOf}(CMM, M) \). Moreover, all models are instances of the core metametamodel: \( \forall M_i \in M : \text{instanceOf}(M_i, CMM) \).

In VPM, metamodels are models which are derived from the core metametamodel, and refine its basic notions to define elements of languages. Practically, a metamodel is both a subtype and an instance of the core metametamodel.

Definition 6 (Metamodel) A metamodel \( MM \) in a model store \( MST \) is a model that is a subtype of the core metametamodel: \( \forall MM \in M : \text{supertypeOf}(MM, CMM) \).

Instance models are models that have a type assignment, expressed by the instantiation relationship. Typing in VPM is a mapping that maps an instance model to a metamodel.

Definition 7 (Instance model) An instance model \( IM \) in a model store \( MST \) is a model that is the instance of a metamodel. Formally, \( \forall IM \exists MM \in M : \text{instanceOf}(IM, MM) \).

These definitions (Def. 6 and Def. 7) allow to introduce an arbitrary number of intermediate metatypes between metamodels and the core metametamodel. Hence, a (direct) type of a metamodel is not necessarily the core metametamodel.

At the bottom of the instantiation hierarchy are a special category of instance models called terminal models. From a practical viewpoint, these are the most common class of instance models that have only types and simply represent data records rather than categories or notions.

Definition 8 (Terminal model) A terminal model \( TM \) in a model store \( MST \) is a model at the bottom of the instantiation hierarchy, i.e. has no instances: \( \forall TM \in M : \neg \exists IM : \text{instanceOf}(IM, TM) \).

Figure 2.2 illustrates the macro structure of the model store as described above. The model store is depicted as a multi-layered structure where the layer boundaries can be dynamically defined, with the exception of the core metametamodel \( CMM \) that lies in the centre, at the top of the inheritance and instantiation hierarchy. Metamodels \( MM_1 - MM_3 \) (on the left) can be freely defined and arranged into an arbitrarily complex supertyping hierarchy. Instance models like \( IM \) can be typed according to multiple metamodels, dynamically (instantiation relationships are shown on the right). Both direct (bold dashed arrows) and indirect (grey dashed arrows) typing relationships are managed according to transitivity rules (Section 2.3.2). Terminal models \( TM \) are shown as the outermost layer of the model store.
2.3. MODELS AND METAMODELS

2.3.2 Transitivity of instantiation and inheritance

**Definition 9 (Direct supertype)** A direct supertype relationship between two models \( M_{super} \) and \( M_{sub} \) exist iff \( \text{supertypeOf}(M_{super}, M_{sub}) \) holds and there is no other model \( M_x \) in the inheritance hierarchy between \( M_{super} \) and \( M_{sub} \): 
\[
\forall M_x : \text{supertypeOf}(M_{super}, M_x) \land \text{supertypeOf}(M_x, M_{sub}).
\]

**Definition 10 (Direct type)** A direct type (instance) relationship between two models \( M_{inst} \) and \( M_{type} \) exist iff \( \text{instanceOf}(M_{inst}, M_{type}) \) holds and there is no other model \( M_x \) in the inheritance hierarchy between \( M_{type} \) and \( M_{inst} \):
\[
\forall M_x : \text{supertypeOf}(M_{type}, M_x) \land \text{instanceOf}(M_{type}, M_x).
\]

**Definition 11 (Transitivity and reflexivity of instantiation and supertyping)** The transitivity rules of instantiation and supertyping are defined formally as follows:

- instantiation is transitive along supertyping:
  \[
  \forall \text{Inst}, \text{Type}, \text{Super} \in M : \text{instanceOf}(\text{Inst}, \text{Type}) \land \text{superTypeOf}(\text{Type}, \text{Super}) \Rightarrow \text{instanceOf}(\text{Inst}, \text{Super})
  \]

- supertyping is transitive:
  \[
  \forall \text{Sub}, \text{Super}_1, \text{Super}_2 \in M : \text{superTypeOf}(\text{Sub}, \text{Super}_1) \land \text{superTypeOf}(\text{Super}_1, \text{Super}_2) \Rightarrow \text{superTypeOf}(\text{Sub}, \text{Super}_2)
  \]

- instantiation is NOT transitive:
  \[
  \forall A, B, C \in M : \text{instanceOf}(A, B) \land \text{instanceOf}(B, C) \not\Rightarrow \text{instanceOf}(A, C)
  \]

- instantiation is irreflexive: \( \forall A \in M : \text{instanceOf}(A, A) \)

2.3.3 Model space

The model store specifies how models are persisted. We use the concept of the model space as a more fine-grained view that shows the internal structure of models (Figure 2.3). By this approach,
the collection of models and their internals is represented as a large, hierarchical graph that can be folded and unfolded (drill down and up) along containment (ownership) links. In the example, the model store containing two models $M_1$ and $M_2$ in a type-instance relationship can be unfolded to reveal the internals of the models. $M_1$, being a metamodel, contains types $Type_1$, $Type_2$ (node types) and $R$ (an edge type), while instance model $M_2$ contains instances $Inst_1$, $Inst_2$ and an unnamed edge of type $R$, respectively.

![Figure 2.3: Model space: a fine-grained view of the model store](image)

This modeling infrastructure (the combination of storage concepts and graphs) is supported by the VPM core metametamodel as shown in Figure 2.4. Models can be represented as graphs by two basic elements: the entity (a generalization of the MOF package, class, or object) and the relation (a generalization of the MOF association end, attribute, link end, slot). An entity represents a basic concept of a (modeling) domain, while a relation represents the relationships between other model elements. Most typically, relations lead between two entities, but the source and/or the target end of relations can also be relations (generally, models). Furthermore, entities may also have an associated value which is a string that contains application-specific data.

**Definition 12 (Model space)** The model space, denoted by $MS = (MST, ME, owner, isAggregation, parent, src, trg, value, multiplicity)$ over a model store $MST$ is a fine-grained, directed, labeled and hierarchical graph representation of models and their contents. The model space consists of entities $E$ (nodes) and relations $R$ (edges) that together make up the set of model elements $ME$. Formally, $ME = E \cup R$.

The predicates and functions are defined as follows:

- $owner : M \times M \rightarrow \text{Boolean}$ expresses that a model is part of another model.
- $isAggregation : R \rightarrow \text{Boolean}$ tells whether the given relation represents an aggregation in a metamodel.
- Functions $src : R \rightarrow E$ and $trg : R \rightarrow E$ correspond to the definition for directed graphs in Def. 1.
- $multiplicity : R \rightarrow \{\text{one-to-one}, \text{one-to-many}, \text{many-to-one}, \text{many-to-many}\}$ maps relations to a one of the four multiplicity labels that impose structural conditions on the model (for a detailed discussion, see Def. 15).
2.4. WELL-FORMEDNESS

In the VPM approach, the declaration and storage of all models, elements, relation configurations and supertype-instance relationships is freely allowed, in order to support the persistence of model stores in both consistent (well-formed) and inconsistent states (inconsistent states can occur e.g. during model manipulation transactions such as editing and transformations). Thus, the VPM model store checks (and not enforces) well-formedness criteria, as summarized in this section.

2.4.1 Supertyping hierarchy

Well-formedness rules are generally defined by type-instance relationships (in the context of multiple models and model elements) and containment (ownership) relationships. For metamodels, the most important well-formedness constraint concerns the inheritance hierarchy: supertyping relationships (as an ordering) form a lattice that represents a hierarchical type system of models [VP03]. This typing hierarchy allows for multiple inheritance but disallows circles along supertyping, with a common ancestor root element at the top of the inheritance hierarchy. This common ancestor is represented by the Model element of the core metametamodel.
2.4.2 Type correctness of models

2.4.2.1 Dynamic and multiple typing

The model store allows the flexible declaration of type-instance relationships by the \textit{instanceOf} predicate, which means that all models and elements can have multiple (direct and indirect) types, or no types at all. Type-instance declarations can be asserted and discarded dynamically (hence, they are not statically computed attributes of objects at creation).

2.4.2.2 Type correctness and ownership

Let $MS$ be a model space with a set $ME$ of model elements over a model store $MST$ of models $M$. Ownership type correctness rules are as follows (Figure 2.5):

- Each instantiation relationship on the model element level (black dashed arrow) requires an instantiation relationship at the model level (grey dashed arrow). Formally, $\forall Inst, T \in ME, M_1, M_2 \in M : owner(Inst, M_1) \land owner(T, M_2) \land instanceOf(Inst, T) \Rightarrow instanceOf(M_1, M_2)$.

- For each IM, MM pair, there is a single (direct) type (in MM) of any instance model element (in IM). Formally, $\forall IM, MM \in M, Inst, T_1, T_2 \in ME : instanceOf(Inst, T_1) \land instanceOf(Inst, T_2) \land owner(Inst, IM) \land owner(T_1, MM) \land owner(T_2, MM) \land instanceOf(IM, MM) \Rightarrow T_1 = T_2$.

![Figure 2.5: Modelspace typing correctness](image)

2.4.3 Type conformance of entities and relations

**Definition 13 (Type conformance of entities)** An entity $E$ is \textit{type conformant} iff all of its contained models and the relations it participates in are type conformant, and there are no parallel relations (sharing both source and target) of the same (direct or indirect) type.

**Definition 14 (Type conformance of relations)** A relation $R$ is \textit{type conformant} iff for all of its types $R_{type}$ it satisfies the following conditions: (i) at its \textit{source end}, the model is a (direct or indirect)
2.4. WELL-FORMEDNESS

instance of \( R_{\text{type}} \)'s source model, and (ii) its target end, the model is a (direct or indirect) instance of \( R_{\text{type}} \)'s target model. Formally, \( \forall R_{\text{type}}, \text{instanceOf}(R, R_{\text{type}}) : \text{instanceOf}(\text{src}(R), \text{src}(R_{\text{type}}) \land \text{instanceOf}(\text{trg}(R), \text{trg}(R_{\text{type}}))). \)

The type conformance condition for relations is illustrated in Figure 2.6. The (unnamed) relation of type \textit{chases} between models \textit{Butch} and \textit{Tom} is type conformant, because its source is a (direct) type of \textit{Dog} (which is the source of its type \textit{chases}), and its target is a (direct) type of \textit{Cat} (which is the target of its type \textit{chases}).

2.4.4 Relation multiplicities

\textbf{Definition 15 (Relation multiplicities)} Relations have \textit{multiplicities}, which impose a restrictions on the model structure. Formally, \( \forall R_{\text{type}}, R_1, R_2 \in R \land R_1 \neq R_2 \land \text{instanceOf}(R_{\text{type}}, R_1) \land \text{instanceOf}(R_{\text{type}}, R_2) : \text{multiplicity}(R_{\text{type}}) \)

\begin{itemize}
  \item \textit{one-to-one}: \( \text{src}(R_1) = \text{src}(R_2) \Leftrightarrow \text{trg}(R_1) = \text{trg}(R_2) \) (two relations of the same type and having the same source have to have the same target and vice-versa)
  \item \textit{one-to-many}: \( \text{trg}(R_1) = \text{trg}(R_2) \Rightarrow \text{src}(R_1) = \text{src}(R_2) \) (two relations of the same type and having the same target cannot come from different sources)
  \item \textit{many-to-one}: \( \text{src}(R_1) = \text{src}(R_2) \Rightarrow \text{trg}(R_1) = \text{trg}(R_2) \) (two relations of the same type and having the same source cannot go to different targets)
  \item \textit{many-to-many}: no such restrictions exist.
\end{itemize}

Note that parallel relations of the same type (sharing both source and target) are not allowed: \( \forall R_1, R_2 \in R : \text{src}(R_1) = \text{src}(R_2) \land \text{trg}(R_1) = \text{trg}(R_2) \Rightarrow R_1 = R_2. \)
2.4.5 Containment and namespaces

Models are arranged into a strict containment hierarchy. Within a container model, each owned model element has a unique local name, but each model element also has a globally unique identifier which is called a fully qualified name (FQN). Fully qualified names are constructed as follows:

- The fully qualified name of a model element is the fully qualified name of its owner and the name of the element concatenated.

- There are top level models with no owner. Their fully qualified names are equal to their names.

The construction of the fully qualified name imposes an important constraint on the VPM modelspace: the containment hierarchy for entities must not contain loops, and for every relation, it must be true that a finite traversal along the source endpoints ends up at an entity (otherwise, the fully qualified name would be infinite). This constraint is enforced by the runtime VPM core implementation. Additionally, all elements have a globally unique ID, which cannot change during the life cycle of the model element (in contrast, names are free to change).

2.4.5.1 Well-formedness restrictions for VPM Containment

The owner function in the VPM model space is constrained formally as follows:

- models cannot own themselves: \( \forall M : \text{owner}(M) \neq M \)

- there is no multiple ownership: \( \forall M^1_C, M^2_C, M_1 \land M_1 \neq M^1_C, M_1 \neq M^2_C : \text{owner}(M_1) = M^1_C \land \text{owner}(M_2) = M^2_C \Rightarrow M^1_C = M^2_C \)

- names are unique among models owned by a container model: \( \forall M_C, M_1, M_2 \land M_1 \neq M_2 : M_C = \text{owner}(M_1) \land M_C = \text{owner}(M_2) \Rightarrow \text{name}(M_1) \neq \text{name}(M_2) \)

- if the isAggregation predicate is true for a relation type, this means that for all its instances, the target element of the relation instance must also be owned by the source element. Formally, \( \forall R_{\text{type}}, R_{\text{instance}} \in R : \text{isAggregation}(R_{\text{type}}) \land \text{instanceOf}(R_{\text{instance}}, R_{\text{type}}) \Rightarrow \text{trg}(R_{\text{instance}}) = \text{owner}(\text{src}(R_{\text{instance}})) \)

2.5 Models and metamodels in VIATRA2

In the VIATRA2 framework, all models can be represented using a textual and a graphical concrete syntax notation. The graphical representation consists of a tree-based view and graph diagrams (as illustrated in Figure 2.11). The concrete textual syntax is referred to as the VIATRA2 Textual Modeling Language (VTML). The syntax of VTML has a certain Prolog flavor, but it offers support for well-founded typing and hierarchical model libraries. This textual format, aside from being an easy-to-learn formalism for programmers, also functions as an input-output facility for automatic model importers and exporters.

2.5.1 Petri net metamodel

Example 3 (Petri net metamodel) A simple Petri net (for Example 1) metamodel, represented in VIATRA2, is shown in Figure 2.7. Basic entity types (Net, Node, Place and Transition in blue, representing the Petri net domain, and String in red, imported from the standard datatypes domain) are represented
by rectangles, while relation types (e.g., nodes, name, id, outArc, inArc) are shown as blue arrows. Values of core predicates (src, trg and name) are shown by black labels (e.g. the VPM name of Net is "Net"). Relation meta-attribute values (multiplicity, isAggregation) are printed between double angle brackets in arrow labels. Supertyping relationships between Place, Transition and Node are visualized using the standard UML generalization notation.

The Petri net metamodel in textual notation is shown in Figure 2.8. This metamodel is a refinement of the VPM core metamodel as all elements of the petrinet namespace are defined by core types: entity and relation. The basic elements of VTML are the following:

- **Unary predicates** declare direct type-instance relationships. For instance, Net is a direct instance of core metatype entity. The parameter specifies the VPM name of the model element.

- **Binary predicates** are used to declare supertyping and relation meta-attributes.

- **Ternary predicates** are used to declare relations where the predicate name represents a direct type-instance relationship, while the first parameter represents the relation name, the second and third parameters refer to the source and target, respectively.

In VTML, ownership is represented by the hierarchical embedding of declarations using curly braces. In predicate declarations, elements can be referenced by their local names or fully qualified names (local names are automatically resolved).

### 2.5.2 Petri net instance model

In VTML, instance models can be defined analogously to metamodels, as the language makes no distinction between models on various metalevels.

An example Petri net instance model is shown in Figure 2.9. In this abstract syntax diagram, types (expressed by the `instanceOf` predicate, as illustrated graphically for Net) are represented by the standard `typename` notation in labels. Entity attribute values (declared using the `value` function) are...
shown between curly braces. Ownership relationships are omitted for the sake of simplicity (with the exception of n0–p1).

The textual representation of the example Petri net instance model is shown in Figure 2.10. Here, the metamodel of Section 2.5.1 is instantiated by using fully qualified name references to the types (e.g. metamodel.net identifies Net). By convention, "unnamed" elements whose VPM names are irrelevant (to the user) are assigned names that start with the "uN" prefix. In this example, helper objects that store attribute values (e.g. names and ids) are stored as such unnamed elements. In VTML, attribute values are declared using the arrow syntax (entity → value).

Finally, Figure 2.11 shows the sample Petri net instance model as edited through the ViATRA2 user interface in Eclipse. In the tree view (on the left), ownership is mapped to tree containment:

- at the top level, the petrinet domain is shown, containing its metamodel and a container model for instance models;
- below petrinet.model, instance model n0 is shown. Entities are decorated with E icons, while relations are decorated with R icons. Labels are rendered with formatting name{value} : type for entities, and name(→ trg) : type for relations.
2.5. MODELS AND METAMODELS IN VIATRA2

Figure 2.9: Petri net instance model example

```java
metamodel.net(n0) {
    datatypes.String(uNn0name) -> "MyNet";
    metamodel.net.name(uNn,n0,uNn0name);

    metamodel.net.place(p0) {
        datatypes.String(uNp0id) -> "P0";
        metamodel.net.node.id(uNi1,p0,uNp0id);
    }
    metamodel.net.nodes(uNa,n0,p0);

    metamodel.net.place(p1) {
        datatypes.String(uNp1id) -> "P1";
        metamodel.net.node.id(uNi2,p1,uNp1id);
    }
    metamodel.net.nodes(uNb,n0,p1);

    metamodel.net.transition(t0) {
        datatypes.String(uNt0id) -> "T0";
        metamodel.net.node.id(uNi2,t0,uNt0id);
    }
    metamodel.net.nodes(uNc,n0,t0);

    metamodel.net.place.outArc(uNoa,p0,t0);
    metamodel.net.transition.inArc(uNia,t0,p1);
}
```

Figure 2.10: Petri net example instance model in VTML

In the graph view (on the right), (part of) the model space is rendered as a flattened graph, without ownership hierarchy. Names of unnamed elements are not shown for clarify.
In modeling scenarios, correspondence modeling is necessary in many modeling applications: for instance, model transformations use correspondence links to indicate mappings between already processed source-target models, and similar links are also frequently used in connecting correlated modeling artefacts across multiple modeling domains (e.g. traceability information between requirements, systems design and platform-specific models of the model-driven architecture).

Correspondence information between models and model elements frequently needs to be represented explicitly as an intrinsic part of the model universe (rather than only using external facilities such as indexes). This is useful since this allows such information to be easily accessed and maintained by model queries and manipulations (e.g. in model transformations). In simple cases, attribute values may be used for this purpose (i.e. in a way similar to foreign keys in relational databases), but the generic approach is to use dedicated traceability metamodels and instance models for this purpose. These may take the form of simple relations as well as association classes (represented by an entity and multiple relations connecting multiple source and target elements).

The flexible nature of the VPM framework provides several convenient ways of defining and using traceability models. Since there is no distinction between metalevels, traceability models may connect models on multiple abstraction levels in a straightforward way.

**Definition 16 (Traceability model)** A traceability model is a (cross-domain) model that contains semantic links between two models. Traceability models can be organised hierarchically and their elements may connect corresponding models and model elements in flexible configurations.

In Figure 2.12, an example of a flexible traceability model hierarchy is shown. In this case, the UML StateChart domain is mapped to Petri nets by the `uml2pn` traceability metamodel. On the higher abstraction level (top of the figure), corresponding instance models `MyCar` and `MyNet` are connected by a relation of type `uml2pn`. The entire model (bottom of the figure) reveals that the traceability model actually consists of several relations (metamodels are shown in purple, while instance models are shown in red):

- type `st2pl` maps UML Statechart States to Petri net Places;
- type `tr2tr` maps UML Statechart Transitionss to Petri net Transitions.
2.7 SUMMARY

This example illustrates how complex model element hierarchies can be represented in a compact way using ownership: entities as well as relations on the top level of abstraction can represent finer-grained structures in arbitrary depth.

2.7 Summary

Overall, the VPM framework allows for building complex model hierarchies in a very flexible way, which makes it easy to adapt to a wide range of modeling scenarios. This is due to two major factors: (i) it features a simple kernel language onto which many advanced concepts can be mapped (such as ordered relationships and multi-valued attributes), and (ii) it allows for multiple inheritance and dynamic multi-typing, which, combined with the fact that well-formedness rules are only checked rather than enforced, means that partially well-formed models (models in an inconsistent state) can be also stored without problems.

In this thesis, the VPM metamodeling language of the VIATRA2 framework [Var04] provides a universal supporting infrastructure for all illustrative examples, as well as an important foundation for the scientific results in domain-specific language engineering (Chapter 3).
Contributions of the current chapter  The current chapter introduces the ViatraDSM framework, which provides the modeling foundations for the results of the thesis (Progress map 1). ViatraDSM is a novel domain-specific language engineering framework that can host multiple modeling languages over a uniform modelspace, with full support for multi-domain modeling (Challenge 1) whereby several different modeling languages can be efficiently and coherently integrated (Contribution 3).
3.1 Domains in modeling languages

Modeling domains are the set of concepts, notions and their relations that represent knowledge within a specialized problem field, such as business process management or embedded medical devices. In model-driven software development, domains are frequently captured by domain-specific modeling languages (DSMLs), which allow domain experts to express their knowledge using high abstraction level formalisms. These domain-specific models may then be integrated into modern agile model-driven software development processes, where they are used - along with general purpose software modeling languages such as UML - for systems design and later on for e.g. code generation.

Domain-specific modeling languages are typically designed to be concise, covering a certain (aspect of the) problem space of an application domain, in order to maximise their usability by non-software-engineer end users. In fact, DSMLs are mostly designed as interrelated language families, that allow a heterogeneous group of domain experts and software engineers to design a complex system from multiple perspectives simultaneously. In the VPM framework, this is facilitated by a flexible modeling infrastructure that allows the unified handling of interrelated system models using multiple DSMLs and provides metamodeling support for sharing models between domains.

Definition 17 (Domain Space) A domain space DS over a model store MST is a labelled view that defines a non-disjunct partitioning scheme of models. A domain is an (modeling) aspect or language that is defined by a metamodel. Formally, $DS = (MST, D, dom, def)$ is a 4-tuple where

- $D$ is set of labels that symbolically represent domains.
- Predicate $dom : M \times D \rightarrow Boolean$ expresses domain membership. Models may belong to multiple domains at the same time.
- Function $def : D \rightarrow M$ defines the metamodel of the domain. Each domain is characterized by a single and unique metamodel.

Domain membership An instance model $IM_1$ is a member of a domain $D_1$ if has at least one model element which is an instance of $D_1$'s metamodel. Formally, $IM_1 \subseteq D_1 \Leftrightarrow \exists ME_1 : owner(ME_1) = IM_1 \land \exists Type : owner(Type) = def(D_1) \land instanceOf(ME_1, Type) \Rightarrow dom(ME_1) = D_1$. Furthermore, the defining metamodel of the domain is also considered to be part of the domain.

Domain-specific views The domains in the domain space represent views of the model space, which means that the heterogeneous collection of models can be enumerated and filtered according to domain membership. Such a view allows the modeler to see only those models that are valid instance models of a language metamodel, and also to see the same model instances from different views as multiple instantiation is allowed.

This is illustrated in Figure 3.1, which shows a configuration of the model store with three domains: $Dom_1$, $Dom_2$ and $Dom_3$ are defined by metamodels $MM_1$, $MM_2$ and $MM_3$, respectively. As instance model $IM$ is an instance of both $MM_1$ and $MM_2$, it belongs to both $Dom_1$ and $Dom_2$ and is therefore visible in both views (depicted as ellipses). As the model store allows for deep instantiation (an arbitrary number of metalevels), metamodel $MM_1$ can be an instance of a more abstract $MM_3$ and thus a member of $Dom_3$. A practical scenario where such modeling flexibility is necessary occurs e.g. when using a multi-layered UML modeling approach, combined with external (non-UML) modeling standards:

- $Dom_3$ may be the domain of UML class models,
3.1. DOMAINS IN MODELING LANGUAGES

• $MM_1$ is an instance of a UML class model (e.g. the UML Profile for Testing and Performance) and at the same time interpreted as a domain-specific metamodel,

• $IM$ is an instance of this metamodel, and at the same time, annotated with types from another domain (Dom$_2$) such as BPMN, as it is directly used in a model for software testing processes.

3.1.1 Domain-specific modeling languages

Definition 18 (Domain-specific modeling language) A domain-specific modeling language $DSML = (AS, \{CS\}, Map, WFR, DS)$ of a domain $Dom$ over a domain space $DS$ is specified by the following language engineering aspects:

• abstract syntax ($AS$), which specifies how logical model elements are stored in the modeling environment;

• the set of concrete syntaxes ($\{CS\}$), which specify the (multiple) textual or graphical representation(s) of models;

• mappings ($Map$) between abstract and concrete syntax representations that specify how abstract syntax models are rendered textually or graphically;

• well-formedness rules ($WFR$), which describe static and language-specific constraints;

• dynamic (operational) semantics ($DS$) which describes the behavioral attributes of the system-under-design.

Definition 19 (Abstract syntax) The abstract syntax of a domain-specific modeling language enumerates the language concepts and their inter-relationships. Formally, it is represented by the “topmost” metamodel $AS$ of all the instance models of the language: $\forall IM : dom(IM, Dom) \land instanceOf(AS, IM) \land \exists MM_x : supertypeOf(MM_x, AS)$.

Definition 20 (Concrete syntax) A concrete syntax of a domain-specific modeling language is a mapping of abstract syntax elements to textual, or graphical representations.
Example 4 (Petri net concrete syntax) An example graphical concrete syntax for Petri nets is shown in Figure 3.2. This concrete syntax (as used in e.g. a Petri net editor) renders "Net" entities as the diagram background, while its contents (places, tokens and transitions) are rendered using custom graphics corresponding to the standard Petri net notation.

Abstract-to-concrete syntax mappings Abstract-to-concrete syntax mappings play an important role in language engineering, as they specify the abstraction level of the graphical or textual notation with respect to abstract syntax models. In practice, abstract syntax models are frequently optimized for e.g. automated processing or minimum redundancy data storage, while a graphical or textual concrete syntax is best if it follows the guidelines and best practices of the presentation layer.

For instance, graphical concrete syntaxes frequently use hierarchies and label abstractions (where structural features of the abstract syntax model, such as the number of elements of a certain type, are shown in the diagrams as labels) to hide unnecessary complexity and to keep the layout of diagrams – even for large models – as compact as possible. Even if the abstract and concrete syntax models are structurally similar (as is, e.g. the case for the Petri net example below), certain aspects of the mapping in-between them have to be supported by some synchronization logic.

Textual notations, on the other hand, are supported by parsers that rely on grammars that represent instance models internally as trees. In such cases, especially if the original abstract syntax of the modeling language does not have a strict tree structure, automated mappings (transformations) are used for abstract-concrete syntax synchronization.

Chapter 6 describes abstract-concrete syntax synchronization using model transformations in detail.

Example 5 (Petri net abstract and concrete syntax mappings) An example illustration of corresponding abstract and concrete syntax representations for Petri nets is shown in Figure 3.3. In this case, there is a one-to-one mapping for most core language concepts, as follows:

- **Places, Tokens and Transitions** are represented by one graphical node each; **Nets** are mapped to the diagram background container.
3.2. THE VIATRADSM FRAMEWORK

3.2.1 Overview

Domain-specific modeling tools are typically constructed and customized based on the needs of a given application domain. The main advantage of this approach is that domain-specific modeling languages are more expressive and precise in their target domain than general-purpose languages, such as UML. Moreover, customized tools are easier to use for domain experts, which is a key advantage since this yields fewer design errors and increased productivity.

Customized tool support, however, can be expensive and time consuming to develop. Designing and implementing a domain-specific editor and the source code generator requires expertise in both the target domain and in development tool design. Thus, to speed up the development process of a new DSML, language engineering frameworks have been developed, that provide high-level automated support for language development. Such tools have facilities that (i) allow the language designer to specify aspects (Def. 18) of a language such as (abstract syntax) meta-model, concrete syntax representations etc. and (ii) create prototype editors based on such languages.

ViatraDSM [35,19,26] is a domain-specific language engineering framework built on the modeling foundations of the VPM framework, with the following main modeling design goals:

1. Rapid prototyping of DSMLs. ViatraDSM allows the language developer to create and modify metamodels and instance models quickly, without the need of going through a complicated (generative) development process on each change to the language (as e.g. EMF [?] does).

2. Rapid prototyping is supported by a syntax-driven editing and visualization interface over the VPM modelspace, which presents domain-specific views and allows the user to instantiate...
and manipulate instance models according to well-formedness rules defined by the language metamodel. Changes to the metamodel are instantly propagated to the user interface by a reflective metamodeling layer architecture.

3. ViatraDSM provides first class support for abstractions and domain integration through flexible yet precise metamodeling foundations for abstract and concrete syntax models organized into a universal inheritance hierarchy.

4. Advanced language engineering by integrated model transformations. Domain-specific language design is not only about metamodeling: DSMLs need to be integrated into model-based development processes by model transformations. ViatraDSM is built on the idea that transformations can also be used within the language engineering framework to provide foundations for various language features. For this purpose, the ViatraDSM framework is tightly integrated with the VIATRA2 model transformation engine, and leverages model transformations to provide high-level support for the evaluation of well-formedness constraints, the interactive execution of dynamic semantics (model simulation), inter-domain mappings, and code generation.

3.2.2 Architecture

The ViatraDSM framework is implemented as an Eclipse [Ecl] plugin, using the model management and transformation features of VIATRA2 transformation framework, and the rendering and editing facilities of the Graphical Modeling Framework [GMF] (GMF) Runtime (Figure 3.4).

![Figure 3.4: The architecture of the ViatraDSM framework](image_url)

As the modeling infrastructure is provided by VIATRA2, all metamodels and models are specified and stored according to the foundations described in Chapter 2; ViatraDSM provides domain-specific views and editors on top of the generic model space. Individual domain-specific views and editors are
3.2. THE VIATRA2DSM FRAMEWORK

integrated into the ViatraDSM framework using the plugin interfaces of Eclipse. As ViatraDSM is primarily aimed at supporting graphical modeling languages, its plugins rely on the GMF Runtime to standard visual diagram editing capabilities such as Properties view, undo-redo support, vector zooming, bitmap and vector graphics export, and printing.

Rapid language prototyping A main design principle of the ViatraDSM framework is to minimize manual coding and to maximize flexibility. All modeling is done reflectively: all (abstract and concrete syntax) metamodels can be edited together with their instances, and instance model editors are provided automatically, so that they adapt to metamodel changes instantly. Well-formedness feedback is provided using standard Eclipse mechanisms (error markers in the Errors view).

ViatraDSM uses a tree-view based representation for abstract syntax instance models (metamodel can be specified and edited using the generic VIATRA2 tree view, see Figure 3.5). In contrast, concrete syntax models are represented both in tree views and graphically on diagrams. Concrete syntax models are stored in a separate modeling layer (with traceability links connecting abstract and concrete syntax models, like in GMF), which allows for complete separation.

Powerful visual abstractions While several existing DSM frameworks provide declarative and automated means for designing a graphical representation for a model element, all of them applies a major restriction, namely, a node in the model is mapped to a node in the graphical representation, and the same applies to edges. Thus, a main goal of ViatraDSM framework is to allow a complete separation of conceptual domain elements from their graphical representation, i.e. to provide arbitrary mappings between domain elements and their graphical representation. This is a complex task supported by model transformations (see Chapter 6 for more details).

Workflow of DSML prototyping The construction of a simple domain-specific visual editor consists of the following steps:

1. Designing the domain metamodel: using a generic model editor, the domain metamodel is defined as a refinement (subtype) of the core metamodel.

2. Designing diagram metamodels: similarly to domain metamodels, diagram metamodels are subtypes of the core diagram metamodel. They describe the structural properties of diagrams (a diagram is essentially a tree structure with node and edge view classes).

3. Implementing view classes: based on the core classes provided by the GMF Runtime Library, simple view classes have to be implemented in Java. A typical view class for a simple model object (such as a UML Class) is about 20-50 lines of Java code.

Transformations In ViatraDSM, VIATRA2 transformations are at the heart of advanced language design. They can be invoked directly from the user interface, and their modifications to meta- and instance models are automatically propagated to the user interface. This allows the language engineer to support complex editing functions, checking of non-trivial well-formedness constraints, code generators etc. with integrated model transformations-based services.

Figure 3.5 shows a screen shot of the framework. This view shows a sample scenario where three domain-specific plugins are being used: Petri nets (top middle), entity-relationship diagrams (top left) and a custom DSML for designing conveyor systems for transportation of airport cargo (bottom left), along with the generic VPM view (bottom middle). In each editor instance, the user can
freely switch between various domains using the domain selector tabs located on the bottom of each editor window. The Outline view shows abstract syntax elements for the currently selected domain (Petri nets), while the Properties view illustrates syntax-guided editing (domain-specific properties are visible only). Syntax-guided editing is also used for diagrams (concrete syntax models) as the Palette of the diagram editor is configured by the domain of the plugin (entity-relationship diagrams in this case).

### 3.2.3 Advanced language engineering aspects

Advanced language engineering aspects in ViatraDSM are supported by model transformations. Specifically, event-driven transformations (Chapter 5) are tailored for interactive usage scenarios where transformations are executed transparently in the background. The rest of the thesis discusses this technology in detail.

**Abstract-concrete syntax synchronization**  As outlined before, ViatraDSM uses separate modeling layers for abstract and concrete syntax instance models. Concrete syntax diagrams are frequently required to present the information contained in abstract syntax models in a more compact, user-friendly way. This poses a challenging synchronization problem, where (i) \( n \)-to-\( m \) mappings need to be maintained in a (ii) bidirectional way (which frequently involves different change propagation...
3.3. DOMAIN INTEGRATION TECHNIQUES

Behavioral semantics  Behavioral (dynamic) semantics, in contrast to the static nature of abstract and concrete syntax specifications, describe how concepts encoded as instance models of the domain-specific language behave and interact with their surroundings. Dynamic semantics can be described either in terms of a well-known formalism such as finite state automata (denotational semantics), or they can be expressed in terms of a simple, core manipulation language (operational semantics). An example for the latter, where the Petri net dynamic semantics is programmed using elementary model transformation operations is shown in Section 4.1.4.5. Based on this approach, a complete design-time simulator is built in Section 7, which allows the user of the domain-specific modeling environment to visually execute and also debug instance models at editing time.

On-the-fly constraint evaluation  In language engineering practice, complex language-specific constraints are formulated in dedicated languages such as the Object Constraint Language (OCL [Obj01]). Complex constraints can also be expressed by graph patterns in the ViATRA² language family, see Section 4.1.2.3 for an example. Incremental model transformations can be used to incrementally check the validity of well-formedness rules and maintain special traceability models that are used to provide feedback to the user (discussed in detail in [20]).

Language mappings and code generation  Language mappings describe how instance models in different domains can be mapped onto each other. In multi-domain modeling, where a complex system is designed using multiple modeling aspects, domains frequently overlap, so consistency has to be ensured. Transformation languages are typically used to express inter-domain relationships, see Section 4.1.3.3 for more details. A special case of transformation-based mappings is code generation, where domain models are mapped into a textual syntax; relevant techniques and a case study are discussed in detail in Section 8.

3.3  Domain integration techniques

3.3.1  Motivation

A main issue that need to be addressed when developing new DSMLs is to reuse existing domain-specific solutions wherever possible since the development of a domain metamodel, which drives the generation of domain-specific editors, can be a time consuming task, and requires a very deep understanding of the specific application domain. Fortunately, many of such metamodels have already been standardized as part of UML profiles (e.g. the UML Profile for Performance, Schedulability and Time [Theg], etc.). These metamodels may also serve as a primary basis for designing full-fledged DSMLs (and not only for tagging UML diagrams by corresponding stereotypes).

On the other hand, UML facilitates the use of multiple views to develop the underlying system model in a modular way according to the separation of concerns principle. However, UML can also be interpreted as a collection of distinct modeling languages (as promoted by the Precise UML group [Theh], for instance), thus the separation of concerns principle imposes further requirements for the integration of DSMLs. For instance, frequent coordination is necessitated between different stakeholders and domain experts (e.g. security experts, software engineer, performance experts, etc.) in a
regular system design process, thus the models captured separately in the different DSMLs need to be kept consistent all the time.

To provide a high degree of coordination between different DSMLs, language engineering frameworks (like Tiger [EEHT05], GME with GReAT [KASS03], VMTS [LLMC04], EuGENia [KRA+10, KPP08]) are complemented with model transformation features to map models of a source DSML into a target DSML in a consistent way. However, a consistent, incremental and bidirectional synchronization between different modeling languages is still a challenging problem.

In this work, we propose a novel approach for multi-domain modeling and integration. First, we propose (in Section 3.3.2) light-weight techniques for integrating multiple domain-specific modeling languages into a consistent system model in addition to traditional model transformation based domain integration. Using these techniques, integration of DSMLs can be specified very easily compared with designing a complex model transformation for the same problem in many cases.

Our approach supports the multi-domain integration of DSMLs in the following ways: (i) subtyping between metamodel elements of two DSMLs, which is an extension of the UML profiling mechanism for an arbitrary host DSML (not only UML); (ii) multiple instantiation (typing) where a model element may be typed over multiple domain metamodels; and (iii) traditional model transformations as provided by the ViATRA2 transformation framework. For all these solutions, the models of the different DSMLs can be edited separately by the domain engineers, however, the underlying system model is kept synchronized. While these different techniques have been proposed previously in a UML context, generalizing them in a DSM context is the main contribution of this work.

Our techniques have been implemented in the ViatraDSM framework (Section 3.4), built on top of the metamodeling and model transformations of ViATRA2.

Domain integration  Complex systems are usually modeled from different perspectives, for example, UML offers several diagram types for software design, e.g. class and deployment diagrams for structural modeling, activity and sequence diagrams for dynamic behaviour etc. Thus the separation of concerns is essential in maintaining the clarity and accuracy of complex models. The various modeling perspectives should be linked together in the underlying system model in a consistent way. This means, in simple terms, that if something is changed in one of the modeling domains (e.g. a method name is changed in a UML class diagram), this change has to be automatically reflected in all other modeling domains (e.g. by changing the name in all sequence diagrams as well). Thus, domain integration means retaining the advantages offered by various domain-specific tools, and providing automated support for generating a global and coherent system model from small domain-specific submodels\(^1\).

The traditional approach to domain integration is to use separate domain-specific tools for various source domains and import their output into the target domain (see Section 3.3.2.1). This is generally unidirectional, i.e. if a change in the target domain should be reflected in the source domain, the only option is to manually figure out what needs to be changed in the source domain and edit the models manually, or to write a second transformation.

These off-line (import/export based) transformational integration approaches can be slow, especially for large models, even if automated tool support is available. A lightweight solution is to integrate all (meta)models into a single language, and use stereotype annotation (tagging) to assign model elements to various domains (Figure 3.6). This approach is used by the profiling mechanism of

---

1 The problem of domain integration is frequently referred to as view integration, especially, in the UML terminology. By our terminology of domain integration, we would like to emphasize that the same problem exists in DSM frameworks as well, although no direct support is provided in most of general purpose DSM tools.
UML. Whilst some tool support exists for defining and applying UML profiles, this technique lacks certain advantages offered by flexible domain-specific editors as it is remarkably easy to construct ill-formed models with inappropriate stereotypes.

![Figure 3.6: Integrated multi-domain modeling](image)

However, the metamodels in standard UML Profiles already incorporate deep knowledge of a domain gathered from top domain experts, which is required for constructing editors and languages for that domain. Unfortunately, these existing metamodels are frequently ignored when constructing a new language in a DSM framework. In our upcoming example, we demonstrate that the metamodels of UML profiles are highly reusable when constructing domain-specific languages.

### 3.3.2 Overview of domain integration techniques

We discuss three techniques for multi-domain modeling integration based on (i) subtyping between metamodel elements of two languages, which is an extension of the UML profiling mechanism for arbitrary host languages (and not only UML); (ii) multiple instantiation (typing) where a model element may be typed over multiple domain metamodels; and (iii) traditional model transformations. In all these cases, the models of the different DSMLs can be constructed separately by the domain engineers, however, the underlying system model is kept integrated automatically by appropriate mechanisms.

#### 3.3.2.1 Transformation-based integration

Transformation-based domain integration (Figure 3.7) means that models from different domains are mapped to each other by model transformation techniques (see Chapter 4 for more details on this technology). In the general case, such a transformation takes a valid model of the source domain, and produces its counterpart in the target domain. The equivalence of the source and target models are ideally ensured by the (formal) verification of the transformation.

The transformation-based integration technique is the most generic approach to multi-domain modeling. Moreover, bidirectional translations are also possible in certain cases. However, for complex models, regenerating the whole target model after a small change in the source model can significantly slow down the design. Therefore, the main challenge of this approach is to provide support for incremental transformations which are more efficient by avoiding the recalculation of previously known and valid facts (e.g. avoiding the regeneration of those parts of the target model which are not
3.3.2.2 Metamodel-level integration by subtyping

However, in many cases, the transformation-based approach can be considered too heavyweight concerning the amount of work required for specifying the links between two domains. This is particularly true when the source and the target domain metamodels are structurally similar. This similarity can be expressed with a (partial) supertypeOf relation between the domain metamodels (Figure 3.8).

This technique is a generalization of UML profiling mechanism for an arbitrary host domain. In Figure 3.8, element tagging means the assignment of an additional supertype to the source domain metamodel’s certain elements. This is possible if the underlying metamodeling framework supports multiple inheritance.

3.3.2.3 Model-level integration by multiple instantiation

The model-level integration approach (see Figure 3.9) requires that a class in target domain can be assigned explicitly as a type for a model element in the source domain. This assignment can be carried out by user interaction or automatically.

The key difference between the model- and metamodel-level approaches is that by mapping metamodels it is ensured that all (most) model instances will be mapped onto the target domain, while model-level mapping is more customizable in the sense that the user decides what to project. A major requirement for model-level domain integration against the underlying metamodeling framework is a support for multiple instantiation, i.e. to allow to assign a type from each domain to an instance-level model element.
3.4 Domain integration over the VPM model space

3.4.1 Multi-aspect system modeling

The VPM metamodeling framework (Chapter 2) provides a flexible modeling environment where all domain concepts and domain integration techniques laid out in this section can be implemented (Section 2.4). First, as supertyping relationships are allowed to encode multiple inheritance (Def. 4), metamodel-level integration can be easily mapped to VPM concepts: (i) domain metamodels are all subtypes of the core metametamodel (Def. 5), but (ii) can freely be organized into an inheritance hierarchy (where they can subtype each other flexibly), as long as well-formedness rules are adhered to (Section 2.4.1). The VPM framework also allows for multiple (direct and indirect) instantiation (Section 2.4.2), and thus fully supports the model-level integration approach.

Well-formedness considerations In domain integration, model instances of a source domain (implicitly or explicitly) become instances of a target domain. However, they are not necessarily well-formed, as some (static and language-specific) constraints may not hold in either of the domains. VPM’s flexible approach to well-formedness checking (where constraint violations are tolerated) allows to implement domain-specific views (Figure 3.1) with increased flexibility compared to strict metamodel-oriented frameworks such as EMF [?], which cannot store partially incorrect models. As the underlying VPM framework tolerates such inconsistencies flexibly, partially consistent models can be persisted and used without problems.
CHAPTER 3. LANGUAGE ENGINEERING IN THE VIATRADM Framework

(a) Types from Domain A are assigned explicitly to model elements in Domain B (either by the user or automatically using a mapping model).

(b) This mapping can be partial, because in most cases not all elements from Domain B are relevant in Domain A.

Figure 3.9: Model-level integration (multiple instantiation)

If models need to be corrected, such corrections can be solved by (simple) model transformations. In contrast to an entirely model transformation-based integration scenario, the transformations required here only perform corrections on the models (instead of full translations).

3.4.1.1 Case study: Enterprise Security Policies and the UML Performance Profile

Figure 3.10 outlines the first demonstrating scenario, which targets to create and maintain two domain-specific views (ESP and UML-Perf) of the same instance models IM within the model store. Light-weight metamodeling techniques will be used for demonstration. This scenario intends to show how a custom domain-specific language can be automatically projected to an analysis domain corresponding to the UML Performance Profile.

Example 6 (The Enterprise Security Policy (ESP) domain) The domain-specific modeling language is built from scratch by using a generic modeling environment. In this case, the Enterprise Security Policy domain (see Figure 3.11, which is an extended version of the example presented in [PW04] as a complex case study) describes a simple enterprise security and surveillance system installation and its security procedures (note: several classes have properties which are omitted in Figure 3.11 for the sake of simplicity).
Figure 3.11 depicts the metamodel for the ESP domain in the textual VTML and a graphical syntax to ease understanding. As a highlight, the domain and model keywords are introduced here; domain declarations are used to indicate the classification of follow-up declarations in terms of the domain space, while the model keyword is used to give a name the particular model that is being defined.

The Enterprise Security Policy domain describes both deployment and procedural aspects of enterprise surveillance systems. At the top of the containment hierarchy is the Installation entity, to which Buildings, a Network and several Processes can be assigned. A Building consists of Rooms, where Devices can be placed. Devices have deployment and operational cost properties which can be used for financial analysis and optimization. The buildings are connected to the network infrastructure; individual devices can be assigned individual connection properties (not shown in Figure 3.11). Security policy Processes consist of Activities, which can make use of several devices.

**Example 7 (UML Performance (UML-Perf) domain)** Since the application domain is capable of describing processes, it is a natural requirement to analyze the performance and throughput of the system in a production environment. For modeling performance related aspects, we use the standard UML Performance profile[Theg]. As a consequence, the modeling environment should be extended to allow the projection of security policy models into the UML Performance domain (Figure 3.12). The UML Performance domain describes Performance Contexts, which consist of Workloads, Performance Scenarios, and Performance Resources. A scenario is made up of multiple, ordered Performance Steps, which can make use of either Passive, or Processing resources. A scenario can be analyzed with either Open, or Closed workloads.
3.4.1.2 Elaboration of the example

To solve the problems outlined in Example 6, we create a multi-view (in ViatraDSM) over the model space consisting of two domain-specific views (corresponding to the ESP and UML performance domains) and show how each present the contents of the same model space specifically to the domain context. Our goal is to enable the performance modeling expert to view the models created by the enterprise security experts directly from the native UML Performance view.

Since the domain metamodels are structurally similar, our approach is focused on the metamodel-level domain integration technique (simple transformations for ensuring well-formedness will only be outlined here, as they will be discussed later in detail: Chapter 4). One could also opt for a model-level integration approach, however, in this specific case primary system design is carried out in the ESP domain, and the UML Performance view is only used for the assignment of performance-specific model properties.

Example 8 (Metamodel-level integration between ViatraDSM’s domains) Figure 3.13 shows the metamodel-level integration mappings in VTML and also graphically. The ESP domain incorporates both structural (Building-Network-Room-Device) and dynamic views (Processes) from UML-Perf. In this case, structural elements (Devices are mapped to (various types of) PResources, while Processes
3.4. DOMAIN INTEGRATION OVER THE VPM MODEL SPACE

```plaintext
domain(UMLPerf);
model(UMLPerf_Meta);

entity(Performance_Context) {
  entity(Workload) {
    entity(Open_Workload);
    supertypeOf(Open_Workload, Workload);
    entity(Closed_Workload);
    supertypeOf(Closed_Workload, Workload);
    relation(scenario, Workload, PScenario);
  }
  entity(PScenario) {
    entity(PStep) {
      relation(pred, PStep, PStep);
      relation(succ, PStep, PStep);
    }
    supertypeOf(PStep, PScenario);
    relation(root, PScenario, PStep);
    relation(host, PScenario, PProcessing_Resource);
  }
  entity(PResource) {
    entity(PProcessing_Resource);
    supertypeOf(PProcessing_Resource, PResource);
    entity(PPassive_Resource);
    supertypeOf(PPassive_Resource, PResource);
  }
}
```

Figure 3.12: UML Performance domain metamodel (extract)

and their Activities correspond to PScenarios and their PSteps.

The mappings are declared by `supertypeOf` clauses in VTML. The metamodel-level mapping ensures that every model instance constructed in the ESP domain will be visible in the UML Performance domain as well (naturally, this holds for the mapped model elements).

Notes on well-formedness  It can be noted that the containment structure of the language metamodels is slightly different (see the green highlight in Figure 3.13). Thus, the containment tree structure of the source domain is not strictly well-formed in the target domain: `Device` instances associated to a given `Installation` can be reached in three navigational steps in the ESP domain, however in the UML Performance Profile `PResources` are directly contained by `Performance Context` instances. This means that for already existing Device instances, such as `LobbyCam` being contained as a valid `PResource` by the `GroceryStore` performance context, a new relation of type `resources` must be added at some point (marked by new keyword in Figure 3.13). This simple correction may be implemented e.g. by a model transformation using graph transformation rules (see Chapter 4 for more details).

Domain-specific views in ViatraDSM  A rendering of the multi-view in the graphical user interface of ViatraDSM is shown in Figure 3.14. The screenshot extracts show the same physical model instance from two different perspectives. As the user is editing the model in the ESP domain, the changes are instantly visible in the UML Performance domain. However, these are domain-specific views, thus in each editor only those details are visible which are relevant for the given visual language.
CHAPTER 3. LANGUAGE ENGINEERING IN THE VIATRADS FRAMEWORK

domain(VPM);
model(UMLPerf__ESP);

/* mapping of structural notions */
supertypeOf(
  ESP.ESP_Meta.Enterprise_Security_Policy.Installation,
  UMLPerf.UMLPerf_Meta.Performance_Context);
supertypeOf(
  UMLPerf.UMLPerf_Meta.Performance_Context.PResource);
supertypeOf(
  UMLPerf.UMLPerf_Meta.Performance_Context.PResource.PProcessingResource);
supertypeOf(
  UMLPerf.UMLPerf_Meta.Performance_Context.PResource.PProcessingResource);
supertypeOf(
  UMLPerf.UMLPerf_Meta.Performance_Context.PResource.PPassiveResource);
supertypeOf(
  UMLPerf.UMLPerf_Meta.Performance_Context.PResource.PPassiveResource);
supertypeOf(
  UMLPerf.UMLPerf_Meta.Performance_Context.PResource.PPassiveResource);
supertypeOf(
  UMLPerf.UMLPerf_Meta.Performance_Context.PResource.PPassiveResource);

/* mapping of dynamic notions */
supertypeOf(
  ESP.ESP_Meta.Enterprise_Security_Policy.Process,
  UMLPerf.UMLPerf_Meta.Performance_Context.PScenario);
supertypeOf(
  UMLPerf.UMLPerf_Meta.Performance_Context.PScenario.PStep);

Figure 3.13: Mapping between the ESP domain and UML performance

3.4.2 Domain integration in language engineering

Multi-domain integration techniques are not only useful in modeling scenarios involving many different languages, but also in designing advanced DSMs. As mentioned in Section 3.2, such (complex) languages are defined by an interconnected hierarchy of metamodels (or, more precisely, their
3.4. DOMAIN INTEGRATION OVER THE VPM MODEL SPACE

(a) UML Performance

(b) Enterprise Security Policy

Figure 3.14: Domain-specific editors: the same model element in two different domains

metamodel is constructed according to a multi-aspect design pattern, where each modeling aspect – sub-language – can be considered a complete metamodel on its own).

In graphical modeling languages, a core multi-aspect language design pattern is to separate the abstract and concrete syntax representations of the language (where, most typically, a single abstract syntax metamodel is accompanied by several concrete syntax metamodels corresponding to different diagram types the language might have). This modeling architecture allows for a high degree of flexibility, since abstract and concrete syntax models can be managed and manipulated according to different considerations: abstract models may be optimized for machine processing or standard compliance, while concrete models may be optimized for usability that relies on powerful abstractions that hide unnecessary detail for conciseness. However, these sub-domains still represent the same logical model, so their consistency has to be ensured. The two crucial factors of enabling the complete, yet consistent separation of abstract and (several) concrete syntax models of DSMLs are (i) a flexible traceability modeling mechanism that allows complex correspondence mappings, and (ii) automated model synchronization that efficiently and transparently maintains all three models.

Figure 3.15 illustrates the modeling foundations of the ViatraDSM framework that implements the previously outlined design pattern. Here, abstract and concrete syntax models are handled as three subdomains \( DSM_{AS} \), \( DSM_{CS} \) and \( DSM_{TR} \), and each consists of an inheritance hierarchy (lattice) as follows:

- **Core metamodels** \( CMM_{AS}, CMM_{CS}, CMM_{TR} \) at the top of the lattice represent the set of basic notions that are common in all DSMLs of the framework. All domain metamodels are refinements of these, and all domain-specific instance models are (indirect) instances, which allows for reflective features, such as generic, tree view-based editors and generic graphical visualizations for diagrams.

- \( CMM_{CS} \) and its subtypes define the structure of diagrams, not their precise appearance, and are used to persist visualization information.

- \( CMM_{TR} \) and its subtypes describe (flexible) mapping rules that are to be followed by well-formed abstract and concrete syntax models. A subtype of \( CMM_{TR} \) is thus a part of a mapping
library that describes how a certain concrete syntax element is to be connected to its abstract syntax counterparts. The information encoded in the instances of traceability models is used throughout the UI, for e.g. collecting both abstract syntax (structure) as well as concrete syntax (visualization) properties in the Properties view.

The mapping rules are enforced by interpretative transformations, which are discussed in detail in Chapter 6. The focus of the current section is to systematically describe the metamodeling foundations of the ViatraDSM framework.

### 3.4.2.1 Abstract and concrete syntax metamodels

Example 9 (Abstract syntax core metamodel) Figure 3.16 shows the core metamodel for abstract syntax (domain) model elements associated to a Hierarchy. A Hierarchy is a top-level model element that represents all instance models of the language. The core abstract syntax metamodel defines a directed, labeled graph with Nodes and Edges; both Nodes and Edges can have Properties. Nodes are organised into a containment hierarchy.

This abstract syntax core metamodel represents the top of the inheritance lattice of all domain-specific languages used by ViatraDSM’s plugins. It conceptually extends the VPM core metamodel (Def. 5) by the explicit representation of user-specifiable property values that can be subtyped to define domain-specific property type systems. The core metamodel was designed so that any domain-specific language can be projected into this simple framework.
Similarly to the abstract syntax metamodel, a concrete syntax (diagram) metamodel may be defined.

**Example 10 (Concrete syntax core metamodel)** Figure 3.17 shows the core metamodel for concrete syntax model elements associated to a Diagram. All elements of a diagram are either NodeFigures or EdgeFigures that can display (textual) Attributes in labels. Diagram elements are freely allowed to contain other diagram elements to express the hierarchical nature of diagrams (where graphical elements are contained by others).

**Petri net examples** We elaborate example projections of the Petri net domain (Example 1) to the core metamodels. In the graphical notation of the following figures, subtyping relationships are implicitly indicated by in brackets (where the supertype is written inside the brackets), for the sake of simplicity.

**Example 11 (Petri net abstract syntax metamodel)** In abstract syntax (shown in Figure 3.18), the topmost nodes that represent distinct instance models are Petri nets, which may contain Places and Transitions. Places may, in turn, contain Tokens. A Token instance is only allowed to be added to a Place instance because of the tokens relation which is a refinement of the core contains concept. OutArcs and InArcs are derived from the Arc core type. Two properties (Place capacity and Arc weight) are explicitly metamodeled.

The VTML code corresponding to the metamodel of Example 11 is shown in Figure 3.19. Here, a top-level supertypeOf clause declares that PetriNet_AbstractSyntax_Meta is a subtype of the core abstract syntax metamodel DSM_AbstractSyntax_CoreMeta (Figure 3.16). This top-level supertype relationship is explicitly elaborated in the rest of the code, as every element subtypes some base notion from the core metamodel.
CHAPTER 3. LANGUAGE ENGINEERING IN THE VIATRADSM FRAMEWORK

Figure 3.17: Core concrete syntax metamodel

Figure 3.18: Abstract syntax metamodel
3.4. DOMAIN INTEGRATION OVER THE VPM MODEL SPACE

We use the same principle for defining the metamodel for standard Petri net diagrams, which show Petri nets, places and transitions but omit tokens for the sake of visual clarity.

Example 12 (Petri net concrete syntax metamodel) In concrete syntax (shown in Figure 3.20), Petri nets, Places and Transitions are mapped to separate visual graph nodes (Petri net root, PlaceFigure and TransitionFigure, respectively), while OutArcs and InArcs are visualised as graph edges (OutArcFigure and InArcFigure). Note that Tokens are not included in the diagram metamodel. Instead of mapping them to separate visual nodes, this visualisation language uses an attribute (TokenCount) to indicate the number of tokens assigned to a place (this is the only label in this simple diagram).

The VTML for the metamodel in Figure 3.20 is defined analogously to Figure 3.19.
3.4.2.2 Traceability models between abstract and concrete syntaxes

To maintain a consistent mapping between instances of the abstract and concrete syntax metamodels, we use representation traceability models (Section 2.6) conforming to a generic traceability metamodel, which is shown in Figure 3.21. It defines a containment hierarchy between TopMappings and MappingElements. A TopMapping instance connects a concrete syntax Diagram with an abstract syntax Hierarchy, so it serves as a top-level container for the rest of the trace metamodel elements. Each DiagramElement (NodeFigures, EdgeFigures) may have multiple MappingElement associations; since MappingElement is abstract, two non-abstract subtypes (EdgeMappingElement and NodeMappingElement) are used to define trace bindings to abstract syntax Edges and Nodes, respectively.

This way, a flexible n-to-m (many-to-many) mapping can be defined, where a concrete syntax element may reference multiple abstract syntax elements by defining multiple MappingElements, or, an abstract syntax element may be connected to multiple MappingElements in the reverse direction. It is important to note that MappingElements are capable of persisting non-structural state information in their VPM values.

To specialize this generic mapping metamodel for the Petri net domain, we follow the design pattern laid out in Figure 3.15 to define a subtype metamodel to encode traceability relationships between the abstract syntax (Figure 3.18 and concrete syntax (Figure 3.20) representations of Petri nets.

Example 13 (Petri net representation traceability metamodel) The refinement of the core metamodel to the Petri net diagram domain is shown in Figure 3.22. In this case, we have constructed a partial one-to-one mapping between the metamodels in Figure 3.18 and 3.20. Note this is still a metamodel, thus it represents a domain-specific variant of the core mapping concepts; also, certain details, e.g. containment relations between PNTopMapping and NodeMappingElements have been omitted from Figure 3.22 for the sake of retaining visual clarity.

Instance model example Figure 3.23 demonstrates how abstract, concrete, and trace metamodels are instantiated to create an interconnected, domain-specific instance model. The VTML syntax is shown in Figure 3.24.

Example 14 (Domain-specific Petri net instance models) In Figure 3.23, the Petri net consists of a place P0 with a token To0, connected by an outarc O0 to a transition T0. This abstract syntax model, as shown with orange, is presented to the user by a tree view (top right). On the diagrams, a
3.4. DOMAIN INTEGRATION OVER THE VPM MODEL SPACE

```plaintext
domain(DSM);
model(DSM_Abstract2Concrete_CoreMeta);

entity(TopMapping) {
  relation(diagram,TopMapping,DSM.DSM_ConcreteSyntax_CoreMeta.Diagram);
  relation(hierarchy,TopMapping,DSM.DSM_AbstractSyntax_CoreMeta.Hierarchy);
  entity(MappingElement) {
    relation(diagramElement,MappingElement,DSM.DSM_ConcreteSyntax_CoreMeta.Diagram.DiagramElement);
    relation(mappings,MappingElement,DSM.DSM_ConcreteSyntax_CoreMeta.Diagram.DiagramElement,MappingElement);
    multiplicity(mappings,one_to_many);
    entity(NodeMappingElement) {
      relation(nodeMapping,NodeMappingElement,DSM.DSM_AbstractSyntax_CoreMeta.Hierarchy.Node);
      multiplicity(nodeMapping,one_to_many);
    }
    supertypeOf(NodeMappingElement,MappingElement);
  }
  supertypeOf(EdgeMappingElement,MappingElement);
  entity(EdgeMappingElement) {
    relation(edgeMapping,EdgeMappingElement,DSM.DSM_AbstractSyntax_CoreMeta.Hierarchy.Edge);
    multiplicity(edgeMapping,one_to_many);
  }
  supertypeOf(EdgeMappingElement,MappingElement);
}
```

**Figure 3.21: The core mapping metamodel**

The core mapping metamodel is used to represent the mapping between the concrete and abstract syntax models. The concrete syntax model (yellow) is rendered where the token count is shown both graphically (bottom right) and in the properties view (bottom left). Not directly visible to the user, the system maintains the trace model (shown in white), which encodes the logical mapping between the two model representations.

**Synchronization driven by traceability models** The PlaceMapping instance stores a reference value of 1, which is used to store the mapped value of the TokenCount attribute. As the user changes this attribute value by entering the new value in the Properties view, the system is expected to react automatically and adjust the number of token instances assigned to P0 in the abstract model (concrete → abstract synchronization). Symmetrically, should the user perform editing directly in the abstract syntax model (through the tree view), e.g. by adding another token to P0, the system should...
keep track of this change and apply the necessary modification to the graphical view by updating the
tokencount attribute value.

The implementation of this synchronization semantics using model transformations is elaborated
in Chapter 6.
3.5 Related work

A number of third-party tools have been developed to simplify the complicated GMF workflow. Exeed [Kol07] and EuGENia [KRPP09, KRA+10] are modern EMF-based domain-specific editor tools of the Epsilon [RPKP08] project. They aid the tool builder in the prototyping phase by significantly simplifying the task of creating an EMF-based reflective tree view editor (in the case of Exeed) or a simple GMF-based graphical editor (EuGENia), by using an annotated textual representation of ECore metamodels.

MetaEdit+ [MEP] is one of the first commercial DSM modeling frameworks with support for
generating and customizing domain-specific editors and code generators. Since 1995, it has been applied successfully in various application domains. While MetaEdit+ supports multiple concrete syntax representations for a conceptual domain model, there is no automated support for working with multiple domains simultaneously within an integrated editor. Model transformations can be implemented by hand-coding using an API. On the MetaEdit+ website, there is an example, where the generated code is “back-annotated”, to enable visual tracking while the debugger is stepping through the code.

Microsoft has also released its DSL tools [Mic] in order to support software factories [Gre]. New domain-specific languages are integrated into Visual Studio as plugins. While Microsoft DSL tools offer an advanced way for customizing the graphical representation, the framework is not based on a model transformation backend which leaves the language engineer with application programming as the only option to create model conversions, code generators or model mappings.

The Pounamu [ZGH04] is a meta-tool for multi-view visual language environment construction for Eclipse-based editors. The tool permits rapid specification of visual notational elements, underlying tool information model requirements, visual editors, the relationship between notational and model elements, and behavioural components. While offering a rich support for the generated editors, model transformations are currently not supported by the framework.

Xactium XMF-Mosaic is an integrated, Eclipse-based extensible development environment for domain-specific visual and textual languages [DAR06]. It has support for domain metamodel specification with OCL constraints, and editor generation (concrete textual and graphical syntaxes can be supported for the same language). Using snapshots, prototyping a new language is accelerated considerably. However, unfortunately, as the generator feature is still in its infancy (e.g. constraints are not propagated to the generated source code), using advanced capabilities of the XMF platform require a considerable amount of manual coding effort.

There are also several DSM frameworks which are complemented with support for model transformations, typically, using a graph transformation [EEKR99] based approach, which approaches show the closest correspondence with our approach.

The TIGER project [EEHT05] (which is a conceptual continuation of the GenGeD [BE00] and DiaGen [KM00] tools from the 90s) primarily aims at generating syntax-directed editors as Eclipse plugins based upon the EMF and GEF technologies and the AGG graph transformation engine. Recent development focused on the tight integration of TIGER’s model transformation infrastructure with Eclipse GMF; currently, TIGER is able to generate GMF-based editors with rich and complex editing facilities (such as the execution of a complex editing action based on a single graph transformation rule). However, currently TIGER only supports the execution of a single graph transformation rule, and thus, lacking a control flow language, complex transformations required for model synchronization can only be implemented using extensive Java-coding.

DiaMeta (a follow-up of DiaGen [KM00]) replaces hypergraph grammars by MOF as provided by the MOFLON tool suite [Min06] to allow users not only to specify domain-specific modeling languages but also to generate corresponding diagram editors. As DiaMeta focuses primarily on freehand editing, their contribution is complementary to ours as our main focus is syntax-driven editing.

The Generic Modeling Environment (GME) [GME] combined with the GReAT model transformation engine [VAS04] provides similar functionality to GMF outside the Eclipse world, with a static one-to-one mapping between abstract and concrete syntax models.

The VMTS [LLMC04] framework also provides support for domain-specific modeling and model transformations by providing plugins for Microsoft Visual Studio. It offers core editing functionalities comparable to Eclipse GEF, but further manual coding is required for visual syntax and editing
functionality. Its distinguishing feature is an optimizing OCL processor for efficient handling well-formedness constraints. Since VMTS includes a powerful model transformation engine, it would be possible to implement an approach that is similar to ours. However, no such research has been published yet.

Most advanced multi-domain modeling features are supported by ATOM3 [dLV02], which defines the concept of view metamodels sharing a common metamodel of a visual language. In [GdL07], user-guided manipulation events are directly represented as model elements in the model store, while triple graph grammars [Sch94] are extended to event driven grammars to determine the kind of event and the model elements affected. Change detection is directly linked to user interface events as this approach primarily targets (domain-specific) modeling environments. Note that this approach, does not rely on live transformations since the transformation context is not preserved; instead, the underlying ATOM3 [dLV02] engine is started whenever an event from the UI is received.

**Summary of comparison** ViatraDSM’s unique features are: (i) it is built on a metamodeling back-end supporting multi-level and multi-view metamodeling (vs. the two-level nature of popular EMF-based tools and most others); (ii) it follows a reflective approach meaning that a DSM editor can be constructed without code generation (vs. the generative approach of EMF-based technologies and Microsoft DSL tools); (iii) it is tightly integrated with a model transformation environment – meaning that model queries and transformations can be used for any kind of design and runtime language engineering feature –, like the front-ends of other transformation tools such as VMTS and ATOM3.

### 3.6 Summary

Domain integration is a practical challenge of importance in model-driven (or model-assisted) software engineering, as domain-specific languages are gaining ground in state-of-the-art development processes. In this chapter, we discussed the key enabling factors of this technology: (i) a flexible modeling infrastructure that can integrate models from a wide range of domains; (ii) light-weight integration techniques that provide mappings without the need for complex transformations; (iii) agile language engineering techniques that enable to build integrated language families; and (iv) integrated transformations to support advanced language engineering needs of powerful and dynamic modeling languages.

We introduced the ViatraDSM framework, built on these foundations, as a novel language engineering environment leveraging the metamodeling and transformation facilities of the Viatra2 framework. The ViatraDSM framework provides the context for the rest of the results presented in this thesis.

The results of this chapter are formulated as thesis contributions as follows: I developed a domain-specific modeling environment, which is based on the integrated model transformation approach. It provides high-level support for the specification of language design aspects (arbitrary abstract-concrete syntax synchronization, static well-formedness checking, model execution/animation and model translations) inside the modeling environment.

2.1 I developed multi-aspect modeling for domain-specific languages through a uniform modelspace [19] (Challenge 1), where both the meta- and instance models for multiple domain-specific modeling languages can be persisted, allowing for viewing the same instance models from different domain-specific perspectives (Section 3.1 and 3.3). I also developed a common mapping
meta-metamodel for visual domain-specific modeling languages, which allows flexible correspondence models between abstract and concrete syntax representations [29,7] (Challenge 2).

2.4 I developed prototype implementations to all of the above results, which are available in the ViatraDSM tool [26] (Section 3.2).

Initial ideas and the first implementation of ViatraDSM were developed in collaboration with Dávid Vágó.
Part II

Event-driven transformations in domain-specific languages
Chapter 4

Preliminaries of model transformations: The \textsc{ViATRA2} approach

4.1 Foundations of model transformations

4.1.1 Basic concepts

Model transformations are a core technology of model-driven software engineering. They facilitate the automated processing and manipulation of models, which is crucial in propagating information along a development toolchain, through various levels of abstraction and representation formats (e.g. from platform-independent, high abstraction-level PIMs to PSMs and finally generated source code, as in the Model Driven Architecture standard by the Object Management Group [MDA01]). While the primary usage scenarios for model transformations are traditionally mappings (inter-domain transformations), they are in fact a versatile tool in various other model-centric software engineering practices such as language engineering and code generation.

Model transformations can be implemented in a number of ways [CH03]: (i) most simply, as conventional programs in a programming language such as Java or C#; (ii) by using a dedicated model transformation tool and its high-level formalism (usually a domain-specific language); finally, (iii) by a combination of native programs and various dedicated technologies (hybrid transformations).

Dedicated model transformation languages of such tools (such as the ATLAS Transformation Language [A. 06] of ATL [ATL], Triple Graph Grammars [KW07], FUJABA [NNZ00], AGG [ERT99], GROOVE [Ren04a], ATOM3 [dLV02], VMTS [LLMC04], Epsilon [RPKP08, KPP08], GrGEN.NET [GBG+06], Xpand and Xtend originating from openArchitectureWare [Ber], to name a few well-known examples) make frequently used patterns in transformations (such as model queries and model-specific manipulation operations) easier to specify, and the underlying execution architecture provides optimizations that are not trivial to implement ad-hoc. The research presented in this thesis has been carried out in the context of our model transformation research centered on the \textsc{ViATRA2} tool.

Model transformations in \textsc{ViATRA2} The \textsc{ViATRA2} model transformation framework [?] is being developed as part of a long-running research project of the Fault-tolerant Systems Research Group of the Department of Measurement and Information Systems at the Budapest University of Technology and Economics, lead by Dr. Dániel Varró. \textsc{ViATRA2} is part of the Eclipse Generative Model Transform-
ers (GMT) subproject, and implements an end-to-end modeling and model transformation toolchain integrated into the Eclipse IDE.

Technically, the VIATRA2 framework provides model transformation facilities over the VPM model space (Chapter 2). As VIATRA2 transformations can flexibly manipulate all aspects of this model space, a wide spectrum of transformation problems are supported: (i) VIATRA2 can execute both in-place and copy transformations (i.e. models can be directly manipulated as well as copied and then altered); (ii) meta-transformations can be formulated in a straightforward way, as type-instance relations can be queried and altered dynamically; (iii) as all models and metamodels are handled uniformly, both unidirectional and bidirectional transformations, as well as complex transformations involving queries and manipulation of multiple meta-levels are feasible.

Figure 4.1: Model transformation concepts

VIATRA2 uses the Viatra Textual Command Language (VTCL) for the specification of model transformations (see Figure 4.1 for an overview of the following concepts). VIATRA2 batch transformations can be compared to batch programs that have an explicit entry point and execute according to traditional control flow elements such as loops and branches, however, the VTCL uses both declarative and imperative elements. Model queries are supported by declarative graph patterns (Section 4.1.2), and model manipulation operations can be specified by declarative graph transformation (GT) rules (Section 4.1.3) or imperative instructions called abstract state machine (ASM) rules (Section 4.1.4). Imperative language constructs are also used to explicitly specify the control flow of model transformations.

VTCL is a full-featured programming language that provides basic facilities such as (optionally typed) local variables and global associative arrays (ASM functions), rules (procedures) with parameter passing (input and output variables), and supports re-use by allowing the interoperation of transformation modules (i.e. all constructs - rules, pattern definitions, global variables - defined in a module can be re-used by other modules).

Relevance Throughout this thesis, all transformation code examples use VTCL (or its extended variants). Thus, in the followings, we briefly overview the features of the language and establish
4.1. FOUNDATIONS OF MODEL TRANSFORMATIONS

model transformation-related conceptual foundations on which the novel contributions of this work are based.

4.1.2 Model queries by graph patterns

Graph patterns (GP) are the atomic units in VIATRA2 for capturing well-formedness rules of a modeling language and, especially, for specifying common patterns used in model transformation rules. Graph patterns are integral part of the VTCL.

Graph patterns represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. A model (i.e. part of the model space) satisfies a graph pattern, if the pattern can be matched to a subgraph of the model using a generalized graph pattern matching technique (presented in [Var04, VHV08, BOR\textsuperscript{+}08]). In this section, we present an informal introduction to graph patterns in VTCL, while their formal semantics is discussed in detail in [VB07].

Definition 21 (Graph pattern) A graph pattern \( GP = (P, V, C, R) \) is a tuple that defines a conjunction of constraints \( C \) over a set of model elements and constant scalars (both identified by parameters \( P \) and internally defined variables \( V \)), and it may re-use other patterns as sub-patterns through references \( R \). The following constraint types may be used:

- **simple unary constraints** define type constraints (value conditions on the instanceOf predicate).
- **simple binary constraints** define value conditions on relation meta-attributes (isAggregation, multiplicity).
- **structural constraints** declare that model elements need to correspond to a certain configuration (as value conditions on src, trg owner and supertypeOf, instanceOf).
- **term constraints** define value conditions on terms constructed from core VPM predicates and functions such as name, value and arithmetic and logical operators.

Model element variables \( v \in V_{me} \) are those variables \( v \) of the graph pattern that directly refer to model elements of the model space (defined by unary type constraints).

Definition 22 (Matches and matching sets of graph patterns) A match \( m \) of a graph pattern \( GP \) is a substitution vector of variable-value pairs that describe value bindings to the variables of the graph pattern. A match is valid over a modelspace \( MS \) if all its model element variables refer to existing elements in the model space: \( MS \models GP \iff \forall v \in V_{me} : m(v) \in MS.ME \). The match set \( \{m\} \) of a graph pattern \( GP \) consists of all possible substitution vectors: \( \forall m \in \{m\} : MS \models GP \).

4.1.2.1 Simple patterns and negative application conditions

Patterns are close to predicates in Prolog, as they have a name, a parameter list, and a body. The body of a simple pattern contains model element and relationship definitions using VTML language constructs. In the top left of Figure 4.2, a simple pattern can be fulfilled by a Petri net place that has at least one assigned token. Here Place\( (P) \) declares a variable \( P \) to store an entity of type Place while Place.tokens\( (R,P,Tok) \) denotes a variable to store a relation of type Place.tokens, which leads from \( P \) to Token \( Tok \). Note that these predicates can be listed in an arbitrary order (unlike in
import petrinet.metamodel;

/* P is a Petri net place with a * token Tok; */

pattern placeToken (P, Tok) =
{
Place (P);
Token (Tok);
Place.tokens(R,P,Tok);
}

/* Tr is a transition without parents * a target place and with non-empty name. */

pattern hasNoTargets (Tr, M) =
{
Transition(Tr);
/* The target place of transition Tr is Place TP. */

neg pattern transitionTarget (Tr,TP) =
{
Place (TP);
Transition.inArc(OA,Tr,TP);
}
check (name(Tr)!="")
}

/* Transition Tr is fireable, iff: */
/* it does not have any empty source places. */
/* In other words: all source places have * at least one token. */

pattern fireable (Tr) =
{
Transition(Tr);

neg pattern notFireable (Tr) =
{
Place(P);
Place.outArc(_,P,Tr);

neg pattern placeWithoutToken (P) =
{
Token(Tok) in P;
Place.tokens(_,P,Tok);
}
}

check (name(Tr)!="")
}

Figure 4.2: Basic patterns, negative patterns and embedded negative patterns

Prolog), i.e. the transformation engine is responsible for the appropriate ordering of predicates by using sophisticated algorithms for search plan generation [VHV08].

The keyword neg denotes if a subpattern serves as a negative condition for another pattern. The negative pattern in the lower left of Figure 4.2 can be satisfied if there is a place (TP) for the transition in the parameter (Tr) that is the target of Tr as indicated by relation OA of type Transition.inArc. If this condition can be satisfied, the outer (positive) pattern matching will fail. Thus the pattern matches to transitions that do not have any target places (sinks).

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

A unique feature of the VTCL pattern language among graph transformation tools is that negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations) when the expressiveness of such patterns converges to first order logic [Ren04b]. This is illustrated in the
lower right of Figure 4.2: here, double negation is used to express universal quantification to state that for a transition to be fireable, all its source places must hold at least one token. Literally, the pattern definition states that there should be no such source place to which no tokens are assigned.

### 4.1.2.2 Pattern calls and OR-patterns

In VTCL, a pattern may call another pattern using the `find` keyword. This feature enables the reuse of existing patterns as a part of a new (more complex) one. The semantics of this reference is similar to that of Prolog clauses: the caller pattern can be fulfilled only if their local constructs can be matched, and if the called (or referenced) pattern is also fulfilled.

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

Pattern calls and alternate (OR) bodies can be used together for the definition of recursive patterns. In a typical recursive pattern, the first body (or bodies) define the halt condition for the recursion, while subsequent bodies contain a recursive call to itself. However, ViATRA2 supports general recursion, i.e. multiple recursive calls are allowed from a pattern. Note that general recursion is not supported by any of the existing graph transformation tools up to now. The example in Figure 4.3 illustrates the usage of recursion.

```vtcl
/* Places P1 and P2 are directly connected by a transition. */
pattern directConnection(P1,P2) =
{
  Place(P1);
  Place(P2);
  Transition(Tr);
  find sourcePlace(Tr,P1); // P1 is a source place of transition Tr
  find targetPlace(Tr,P2); // P2 is a target place of transition Tr
}
/* Place P2 is reachable from P1 through a (sequence of) transition(s). */
pattern reachable(P1,P2) =
{
  find directConnection(P1,P2);
} or {
  Place(P1);
  Place(P2);
  Place(MiddleMan);
  find directConnection(P1,MiddleMan);
  find reachable(MiddleMan,P2);
}
```

Figure 4.3: Recursive patterns

In Figure 4.3, a place P2 is reachable from another place P1, if they are directly connected by a transition (Tr), or, it is directly connected to a third place MiddleMan from which P2 is reachable. The pattern uses recursion for traversing inter-place connections through transitions, and uses multiple bodies to create a halt condition (base case) for the recursion (transitive closure).

### 4.1.2.3 Pattern cardinalities

Pattern calls may be extended with a cardinality constraint to count how many ways they can match, using the following syntax: `find somePattern(PassedParameter, QuantifiedParameter,`
... # Count. Here, the variable Count represents the integer value of how many ways the quantified variables can be substituted to form a match of somePattern, while some other parameters of the called pattern may be bound to variables of this match of the calling pattern. Quantified variables are exactly those that do not appear anywhere else in the pattern, just like the quantified variables of a negative pattern call. In fact, the negative pattern call can be regarded as a special case of match counting, with the Count bound to zero. Non-quantified variables have to be defined by at least one pattern construct (entity, relation, positive pattern call); it is not allowed for them to only appear in other match set counters, negative pattern calls, check expressions, or as a symbolic parameter.

```
pattern capacityViolation(P) =
{
  Place(P);
  CapacityConstraint(CC);
  Place.capacity(_,P,CC);
  find placeToken(P,Tok) # Toks;
  check(Toks > value(CC));
}
```

Figure 4.4: Petri net capacity constraint expressed as a graph pattern using cardinality

Such graph patterns are suitable for expressing well-formedness constraints of domain-specific modeling languages (Section 3.1.1).

Example 15 (Petri net capacity constraint) A widely used extension of Petri nets defines capacity constraints for places, which impose an upper limit on the number of tokens that can be assigned to a place at any time. A graph pattern-based definition of this constraint is shown in Figure 4.4, where the violations of these constraint are described. Here, the capacity constraint assigned to place $P$ is stored in an entity $CC$ of type CapacityConstraint. The actual number of tokens assigned is represented by the cardinality $Toks$ of pattern placeToken (with a bound parameter $P$). The constraint is violated if $Toks > value(CC)$.

4.1.2.4 Generic patterns

To provide algorithm-level reuse for common transformation algorithms independent of a certain metamodel, VIATRA2 supports generic and meta-transformations, which are built on explicit instance-of relations of the VPM metamodeling framework.

For instance, we may generalize domain-specific connection patterns (e.g. sourcePlace) as a generic connection between two model entities. The generic example of Figure 4.5 introduces the generic pattern connectedBy which takes an additional type parameter ($RelType$) to express that $Src$ and $Trg$ are connected by a relationship of this type. Type parameters can be bound to arbitrary values during execution time.

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

Note that while other GT tools may also store metamodels and models uniformly in a common graph structure, only PROGRES [SWZ99] supports type parameters in rules, while none of them supports manipulation of (existing) type-instance relationship like the dynamic reclassification (re-typing) of objects. In VIATRA2, generic algorithms (e.g. transitive closure, graph traversals, fault modeling etc.) can be reused without changes for different metamodels.

4.1.3 Graph transformation rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules [EEKR99], which heavily rely on graph patterns for defining the application criteria of transformation steps. The application of a GT rule on a given model replaces an image of its left-hand side (LHS) pattern with an image of its right-hand side (RHS) pattern. The VTCL language support the definition of graph transformation rules using the gtrule keyword.

Definition 23 (Graph transformation rule) A graph transformation rule \( GT = (Input, Output, Pre, Post, Action) \) is a tuple that defines a complex model manipulation operation that takes input parameters \( Input \) and produces output through output parameters \( Output \). Graph transformation rules are defined in terms of two graph patterns and an optional sequence of elementary operations:

- the precondition pattern \( Pre \) describes a condition on the model that has to be satisfied for the rule to be executable;
• the postcondition pattern $Post$ describes a condition on the model that has to be fulfilled after the rule has been executed;

• the optional sequence of elementary operations $Action$ lists operations that have to be executed after the rule has been executed.

4.1.3.1 Specification of graph transformation rules

The sample graph transformation rule in Fig. 4.6 defines a sub-step of the execution of Petri nets, where a new token is added to a place.

![Figure 4.6: Sample graph transformation rule](image)

The syntax of a GT rule corresponds to the traditional notation (see the code in Fig. 4.7). It contains a precondition pattern for the LHS, and a postcondition pattern that defines the RHS of the rule. In general, elements that are present only in (the image of) the LHS are deleted, elements that are present only in RHS are created, and other model elements remain unchanged. Moreover, further actions can be initiated by calling any ASM rules within the action part of a GT rule, e.g. to report debug information or to generate code. This action part is executed after the model manipulation part is carried out according to the difference of the precondition and postcondition part.

```vcl
class PetriNetModel {
    // Add a new token to a place P in net N.
    gtrule addToken(inout N, inout P, out Tok) = {
        precondition lhs(N, P) = {
            Net(N);
            Net.nodes(_,N,P);
            Place(P);
        }
        postcondition rhs(P, Tok) = {
            Place(P);
            Token(Tok);
            Place.tokens(_,P,Tok);
        }
    }
}
```

Figure 4.7: Graph transformation rule in VTCL

Negative conditions are also commonly used in precondition patterns, especially, for model transformations between modeling languages in order to prevent the application of a GT rule twice on the same matching. Negative conditions in the RHS are ignored.
4.1.3.2 Semantics of graph transformation rules

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

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

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

The postcondition of a GT rule may prescribe three different operations on the model space.

- Preservation. If an input parameter of the postcondition appears in the pattern body, then the matching model element is preserved. The element matched by variable $P$ above is thus preserved.

- Deletion. If an input parameter of the postcondition does not appear in the body of the postcondition pattern then the corresponding model element is deleted.

- Creation. If a variable which appears in the body of the postcondition pattern is not an input parameter of the postcondition, then a new model element is created, and the variable is bound to this new model element. In our example above, variable $Tok$ is not an input parameter of the postcondition, thus it prescribes the creation of a new Token element in $P$. This $Tok$ is an output parameter of the postcondition and it is also passed back to the GT rule itself.

Pattern calls in GT rules. In order to reduce the size and to support the modular creation of GT rules, pattern calls (i.e. the find construct) are allowed in both the precondition and postcondition pattern. Its use in the precondition pattern was already discussed in Section 4.1.2.2. ViATRA2 handles non-recursive and non-negative calls in the postcondition pattern, which allows a macro-like substitution of the called pattern in the body of the postcondition. This way, repetitive parts of a postcondition can be modularized into predefined patterns, which can be used in various GT rules afterwards.

4.1.3.3 Cross-domain mappings with graph transformation rules

It is important to emphasize that graph transformation rules can be also used to express cross-domain mappings between multiple modeling languages. In a typical case of a mapping between a source and target modeling language, graph transformation rules contain:

- a semantic model context of the source language in the precondition. This context represents a basic unit of the language which can be mapped to the target language.

- the corresponding model configuration of the target language in the postcondition.
• Traceability links (according to Section 2.6) are used (i) in the postcondition to connect corresponding model elements and (ii) in the negative application condition of the precondition to ensure that this rule will not execute for already mapped model contexts.

![Figure 4.8: UML Statechart to Petri net mapping sample graph transformation rule](image)

**Example 16 (UML Statechart to Petri net mappings)** An example mapping rule between the UML Statechart and Petri net domains is shown in Figure 4.8. Here, a combination of two connected statechart states $S_1$ and $S_2$ and a transition $T$ between them is mapped to a combination of two Petri net places $P_1$ and $P_2$ and a transition $Tr$.

### 4.1.4 Control flow

To control the execution order and mode of graph transformations, VTCL includes language constructs that support the definition of complex control flow. VTCL relies on a basic set of Abstract State Machine (ASM) language constructs [BS03] that correspond to the constructs of conventional programming languages.

**Definition 24 (ASM Rule)** An ASM Rule $ASM = (Input, Output, RuleBody)$ is a tuple that defines a combination of elementary rule invocations ($RuleBody$) that takes input parameters $Input$ and produces output through output parameters $Output$. Elementary rule invocations may be the following:

- **data manipulation rules** (let, update) may be used for the declaration and updating of variables;
- **control flow rules** (if-then-else, try, call, seq, random, iterate) may be used to organize elementary rule invocations into control structures;
- **elementary model manipulation rules** (new, del, rename, move, copy) may be used for simple manipulation operations on the model space;
- **model query and graph transformation invocation rules** (forall, choose) may be used to retrieve information from model queries and invoke graph transformation rules;
- **native rules** may be used to invoke programs implemented in Java.

In the followings, we give an introductory overview of each of these built-in rules.
4.1. FOUNDATIONS OF MODEL TRANSFORMATIONS

4.1.4.1 Basic ASM rules

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

The core set of basic ASM rules are overviewed in Figure 4.9.

```plaintext
// variable definition
let X = 1 in ruleA

// variable (and ASM function) updates
update X = X + 1;

// print and log rules print a term to standard output or into the log
print("Print X: " + X + ":\n");

log(info, "Log X: " + X);

// conditional branching by a logical condition or by pattern matching
if (X > 1) ruleA else ruleB
if (find myPattern(X)) ruleA else ruleB

// exception handling: rule2 is executed only if rule1 fails
try rule1 else rule2

// calls the user defined ASM rule myRule with actual parameter X
call myRule(X)

// the sequencing operator: executes its subrules in the given order;
seq { rule1 ; rule2 ; }

// executes a non-deterministically selected rule from a set of rules
random { rule1; rule2; }

// iterative execution by applying rule1 as long as possible
iterate rule1;
```

Figure 4.9: Overview of core ASM rules in ViATRA2

ASMs provide control structures including the sequencing operator (seq), rule calls to other ASM rules (call), variable declarations and updates (let and update constructs) and if-then-else structures, non-deterministically selected (random) and executed rules (choose), iterative execution (applying a rule as long as possible iterate).

4.1.4.2 Model manipulation rules

In addition to these core ASM rules, the ViATRA2 dialect of ASMs also includes built-in rules for manipulating the model space. As a result, elementary model transformation steps can be specified either in a declarative way (by graph transformation rules) or in an imperative way by built-in model manipulation rules. Main model manipulation rules are summarized in Figure 4.10.

Most of these rules are rather straightforward, we only give more insight into the copy and move rules. The copy rule aims at copying an entity and all the recursive contents (i.e. the subtree of the entity) to a new parent. ViATRA2 provides two kinds of semantics for that copy operation: keep_edges and drop_edges. In the first case, all relations leading out from or into an entity placed anywhere below the copied entity are copied as well. In the latter case, only those relations are copied where both the source and the target entity are below the copied entity. In case of the move rule, an entity is moved (by force) to a new container by decoupling it from its old container. While this step may break the invariants of the old container, this problem is not as critical as in case of EMF as our constraints are only optionally checked (typically at the end of the transformation).
4.1.4.3 Invoking model queries and graph transformations

Model queries may be specified by using simple terms (constructed using arithmetic and logic operators with basic predicate values - such as name, value - and ASM functions), or calling graph patterns (using the find keyword). Graph patterns may be used to retrieve one (randomly selected) match (choose) or all matches (forall). Model queries require a (list of) quantified (running) variable(s) that may be augmented with scopes (in/below) that will be bound by the pattern matcher; the result of the query will be available in these variables in subsequent rule calls (for examples, see the first part of Figure 4.11).

Similarly to model queries, graph transformation rules can also be invoked from ASM programs. The basic invocation of a graph transformation rule is initiated using the apply keyword within a choose or a forall construct. In each case, the actual parameter list of the transformation has to contain a valid value for all input parameters, and an unbound variable for all output parameters. A rule can be executed for all possible matches (in a single, pseudo-parallel step) by quantifying some of the parameters using the forall construct. Finally, a GT rule can be applied as long as possible by combining the iterate and the choose constructs. The example in the bottom half of Figure 4.11 illustrates some possible invocations of our sample rule addToken.

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

4.1.4.4 Native functions and rules

VIATRA2 supports the usage of native functions and rules, which are only declared in the VTCL code, but implemented using Java. These methods can access any Java library (including database access, network functions, and so on), and also the model space. This allows the implementation of complex calculations during the execution of model transformations.
4.1. FOUNDATIONS OF MODEL TRANSFORMATIONS

// myAsmFun is declared to be a unary associative array
asmfunction myAsmFun/1;

// executes rule1 for a (non-deterministic) substitution of variable X
// which satisfies the pattern (or location) condition with X
choose X below M with (myAsmFun(X) > 0) do rule1
choose X below M with find myPattern(X) do rule1
// pseudo-parallel execution of rule1 for all substitution of variable X
forall X below M with (myAsmFun(X) > 0) do rule1
forall X below M with find myPattern(X) do rule1

// execution of a GT rule for one (randomly selected) place P
// variable N needs to be bound
// variable Tok must not be bound
choose P apply addToken(N, P, Tok);

// calling the rule for places
forall P do apply addToken(N, P, Tok);

// calling the rule for all possible matches in parallel
// variables need not be bound
forall N, P do apply addToken(N, P, Tok);

// apply a GT rule as long as possible for the entire model space
iterate choose N, P apply addToken(N, P, Tok)

// myNativeRule is declared to be a native function
@native
rule myNativeRule(in X);

@native
asmfunction myNativeFunction/1;

@native
rule testRule(in X) = seq
{
  if (myNativeFunction(X) % 3 == 0) print("A");
  else call myNativeRule(X);
}

Native code can be used using two language facilities:

- **native rules** are declared by extending their signatures with the `@native` annotation. These rules can be invoked (with full support for parameter passing) with the `call` keyword, just like normal rules.

- **native functions** are declared similarly to native rules using the `@native` annotation and can be used inside terms to evaluate special conditions on scalars or model elements (values that can be passed in variables). They can return scalars (booleans, strings etc.) as well as model element references.

See Figure 4.12 for an illustration of native functions and rules.
CHAPTER 4. PRELIMINARIES OF MODEL TRANSFORMATIONS

4.1.4.5 Complex transformations

These basic built-in ASM rules, combined with graph patterns and graph transformation rules, form an expressive, easy-to-use, yet mathematically precise language where the semantics of graph transformation rules are also given as ASM programs (see [Var04] for more details). The following example (in Fig. 4.13) demonstrates a simple Petri net simulator, written using the examples (and some extensions omitted for simplicity) of the current section of the thesis.

```
// global array, will be used to store the number of firings
asmfunction counter /1;

// performs the firing of transition T
// according to the dynamic semantics of simple Petri nets
rule fireTransition (in T) = seq
{
  // remove tokens from source places
  forall Place with find sourcePlace(T,Place) do apply removeToken(Place);

  // add tokens to target places
  forall Place with find targetPlace(T,Place) do apply addToken(Place);

  // increment the firing counter
  update counter("firings") = counter("firings")+1;
}

// simple Petri net simulation program
// randomly executes firings in the Petri net designated by PN
rule main (in PN) = seq
{
  // initialize counter value for statistics
  update counter("firings") = 0;

  // start an iteration loop
  iterate seq
  {
    // exit condition from the iterate loop: maximum number of firings
    if (counter("firings") > 1000) fail;

    // fire a randomly selected fireable transition
    choose T with find fireable(PN,T) do call fireTransition(T);

    // print a log message to the output
    log("Fired " + counter("firings") + " times.");
  }
}
```

Figure 4.13: A sample ASM program driving a model transformation

**Example 17 (Petri net dynamic semantics)** This program encodes an *in-place, intra-domain* model transformation where an elementary step performs the firing of a Petri net transition according to the simple dynamic semantics of the language (Example 2):

- a transition is *fireable*, if all its source places have at least one token (expressed by the `fireable` pattern in Figure 4.2);

- a *firing* consists of removing one token from each of its source places and adding a token to each of its target places (rule `fireTransition` in Figure 4.13).

In this example, the main rule calls the elementary simulation step in an iterate loop, for a randomly selected fireable transition until the maximum number of firings is reached.
4.2 Incremental graph pattern matching

In order to support incremental graph pattern matching, the ViATRA2 framework implements and adapts the RETE-approach [For82] to support the transformation language of the ViATRA2 model transformation framework (VTCL). Since this approach provides full support for the rich language constructs of ViATRA2, it significantly supersedes and extends the first (and relatively old) RETE-based graph transformation approach [BGT91].

4.2.1 Core idea

In case of incremental pattern matching, the occurrences of a pattern are readily available at any time, and they are incrementally updated whenever changes are made. As pattern occurrences are stored, they can be retrieved in constant time\(^1\), making pattern matching a very efficient process. Besides memory consumption, the drawback is that these stored result sets have to be continuously maintained, imposing an overhead on update operations.

Incremental graph pattern matching and incrementality of transformations  In model transformations, pattern matching is used for model queries, e.g. to find the occurrences of left-hand side (LHS) patterns of graph transformation rules. Incremental transformations (Section 5.1) require that changes to model are propagated in an incremental way, in order to restrict complex calculations to those parts of the model that are affected by an (evolutionary) change. Incremental pattern matching, as introduced in the current section, plays a central role in our approach to incremental transformation technology, as it enables the precise detection of model changes, as well as provides a high performance architecture for the incremental evaluation of model queries.

Since pattern matching can be an important performance factor, such an incremental approach may lead to better performance, especially when transformations are increasingly matching-intensive instead of being manipulation-intensive. In this section, we introduce an incremental pattern matcher component for the ViATRA2 framework; our technology is based on the RETE algorithm [For82], which is a well-known technique in the field of rule-based systems.

4.2.2 Workflow

Initialising an incremental pattern matching engine involves the following conceptional steps:

1. The transformation designer defines various patterns and transformation rules.

2. An incremental pattern matcher (in our case, a RETE network) is constructed based on the pattern definitions.

3. The underlying model is loaded into the incremental pattern matcher as the the initial set of matches.

Typically Step 2 and 3 are carried out in RETE networks a single, interleaving process (as to be discussed in Section 4.2.4.8). Furthermore, the initialization need not be complete; the pattern matcher RETE network can be freely extended (on demand) with additional patterns at a later phase. It is worth pointing out that a RETE-based incremental pattern matcher can be integrated with any a graph transformation engine or any other underlying model manipulation library. For instance, a

\(^1\)excluding the linear cost induced by the size of the result set itself
GT engine with a RETE-based incremental pattern matcher necessitates the repeated execution of the following steps (see Figure 4.14 for illustration):

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

![Figure 4.14: Incremental pattern matching information flow](image)

### 4.2.3 Architecture

Since the ViATRA2 model transformation framework is designed in a way such that it is extensible with alternative pattern matcher modules, our implementation of a RETE-based matcher is based on this as illustrated on Figure 4.15. The incremental pattern matcher offers (implements) the standard pattern matcher interface, and while the RETE network is being constructed, it loads the contents of the initial model. The key architectural difference from the standard local search-based pattern matcher is that the incremental pattern matcher subscribes for change notifications from the model space; this allows RETE to update the results sets automatically whenever changes are made to the model.

### 4.2.4 Adapting the RETE algorithm for ViATRA2 transformations

The RETE algorithm, introduced in [For82], has a wide range of interpretations and implementations. This section describes how we adapted the concepts of RETE networks to implement the rich language features the ViATRA2 graph transformation framework. In this section, we will gradually construct a RETE-based pattern matcher capable of matching the pattern isTransitionFireable, that logically corresponds to the fireable transition graph pattern in Figure 4.2. Here, an extended variant of the pattern definition is used, that also takes inhibitor arcs into account, and uses pattern calls to demonstrate the combination hierarchy of RETE networks.
4.2. INCREMENTAL GRAPH PATTERN MATCHING

4.2.4.1 Tuples and Nodes

The main ideas behind the incremental pattern matcher are conceptually similar to relational algebra. Information is represented by a tuple consisting of model elements. Each node in the RETE net is associated with a (partial) pattern and stores the set of tuples that conform to the pattern. This set of tuples is in analogy with the relation concept of relational algebra.

The input nodes are a special class of nodes that serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing unary tuples representing the instances that conform to the type. Similarly, there is an input node for each relation type, containing ternary tuples with source and target in addition to the identifier of the edge instance. Miscellaneous input nodes represent containment, generic type information, and other relationship between model elements.

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

4.2.4.2 Joining

The key component of a RETE is the join node, created as the child of two parent nodes, that each have an outgoing RETE edge leading to the join node. The role of the join node can be best explained with the relational algebra analogy: it performs a natural join on the relations represented by its parent nodes.

Figure 4.16 shows a simple pattern matcher built for the sourcePlace pattern illustrating the use of join nodes. By joining three input nodes, this sample RETE net enforces two entity type constraints and an edge (connectivity) constraint, to find pairs of Places and Transitions connected by an out-arc.
4.2.4.3 Updates after model changes

The primary goal of the RETE net is to provide incremental pattern matching. To achieve this, input nodes receive notifications about changes on the model, regardless whether the model was changed programmatically (i.e. by executing a transformation) or by user interface events.

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

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

Figure 4.17a shows how the network in Figure 4.16 reacts on a newly inserted out-arc. The input node for the relation type representing the arc releases an update token. The join node receives this token, and uses an effective index structure to check whether matching tuples (in this case: places) from the other parent node exist. If they do then a new token is propagated on the outgoing edge for each of them, representing a new instance of the partial pattern "place with outgoing arc". Figure 4.17b shows the update reaching the second update node, which matches the new tuple against those contained by the other parent (in this case: transitions). If matches are found, they are propagated further to the production node.
4.2. INCREMENTAL GRAPH PATTERN MATCHING

4.2.4.4 Pattern Call

An important feature of the RETE algorithm is that network parts can be shared between patterns, thus reducing space and time complexity. The transformation designer may decompose patterns into smaller, reusable parts calling each other (also called pattern composition).

When a pattern calls another pattern, it can simply use the appropriate production node to obtain the set of tuples conforming to the other pattern. Naturally, the production node may have children attached like any other nodes. It is even possible to define recursive patterns that call themselves; in such cases, the production node of the pattern will have an edge leading back to one of the previous nodes. It is the designer’s responsibility to ensure that the recursion is well-founded and that there is always exactly one fixpoint as result.

Figure 4.18a shows the matcher for pattern isInhibited provided that the simple patterns placeNonEmpty and sourcePlaceInhibitor already have their respective matchers constructed. The matcher selects tuples where the corresponding transition is inhibited by the place for whom the place inhibits the transition, and the place has at least one token.

4.2.4.5 Negative Application Conditions

A powerful feature of VIATRA2 is to embed patterns into each other as negative application conditions, thus allowing negation at arbitrary depth (Section 4.1.2.1). To support such negative pattern calls, the existing mechanism for pattern calls can be used, but the production node has to be connected to a negative node instead of a join node. A negative node (in the RETE network) has two distinct parents: primary and secondary inputs, respectively. The negative node contains the set of tuples that are also contained by the primary input, but do not match any tuple from the secondary input (which corresponds to antijoins in relational databases, see a similar idea with left outer joins e.g. in [VFV05]).
CHAPTER 4. PRELIMINARIES OF MODEL TRANSFORMATIONS

pattern sourcePlaceInhibitor(T,P) = {
    Transition(T);
    Place(P);
    Place.inhibitorArc(IHA,P,T);
}

pattern placeNonEmpty(P) = {
    Place(P);
    Token(Tok);
    Place.tokens(_,P,Tok);
}

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

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

(a) isInhibited  
(b) notEnabled

Figure 4.18: Positive and negative pattern calls

Figure 4.18b shows the matcher for pattern notEnabled, provided that the simple patterns placeNonEmpty and sourcePlace already have their respective matchers constructed. The matcher selects the transitions with source places that do not have any tokens.

4.2.4.6 Disjunction

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

Figure 4.19 shows the matcher for pattern isTransitionFireable (which is an extension of the original definition in Figure 4.2), containing an inline negative pattern with two bodies. In this case, each
4.2. INCREMENTAL GRAPH PATTERN MATCHING

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

Figure 4.19: RETE matcher for the isTransitionFireable pattern

body is a simple reference to a previously constructed pattern, connected to a single production node for the inline pattern.

4.2.4.7 Term Evaluation

In addition to simple graph-based structural constraints, the ViATRA2 framework supports the use of attribute conditions to restrict the names and values of model elements. Various arithmetical and logical functions, or even user-provided arbitrary Java code can be applied to model elements to check the validity of a pattern.

The term evaluator node propagates only those tuples that pass a given test. Furthermore, it registers the affected elements of incoming tuples (regardless whether they had passed the filter or not), so that whenever one of these elements experience change, the tuples containing it can be re-evaluated. If the result changes, the appropriate update tokens will be propagated. The node will monitor changes influencing a tuple until that tuple is finally removed by a negative update received from the parent node.

4.2.4.8 Construction

Given the definition of a pattern, the method to construct a RETE net for finding the matches of a pattern with good efficiency is a non-trivial task. The heuristics employed by ViATRA2 is a straight-
forward, but not necessarily optimal approach.

The key is perceiving a pattern as a collection of constraints imposed on subsets of the group of pattern variables. The construction algorithm processes these constraints one by one, and continues a connected sequence of nodes (“the line”) to match larger and larger partial patterns, eventually using up all constraints and connecting the last node to the production node.

For simple entity and type constraints, pattern calls and miscellaneous cases (e.g. containment), (1) the appropriate input node or production node is accessed; (2) a join node will be attached as a child to it and also to the end of the line; (3) the join node will be prepared to match against variables that are involved in the constraint and are already introduced in the line. For negative application conditions, a negative node is used instead of the join node in an otherwise similar setup. A different setup is required for check conditions (and some miscellaneous cases including injectivity constraints), where a single filtering node (in this case, a term evaluator node) is attached at the end of the line.

When a child node is connected, it automatically receives all the tuples stored by the parent node as positive update tokens (and becomes subscribed for further updates); this way the construction and loading of the RETE net happens simultaneously, even though they are conceptionally separate. Input nodes and production nodes of called patterns are created upon first access; for production nodes, the matcher of the called pattern is also built at this time. This on-demand behaviour ensures that no unnecessary network parts are built and no unnecessary update notifications are delivered. The systems also supports extending an already built and used RETE network with new matchers if the need for new patterns arises.

4.3 Summary

In this chapter, we established the foundations for the ViATRA2 model transformation technology that provides the framework for the contributions of the thesis. We outlined the core concepts and syntax of the ViATRA2 Textual Command Language (VTCL) that is used throughout this work for transformation code examples. We also described the incremental graph pattern matcher of ViATRA2, which provides the technological foundations for the novel results of Chapter 5.

In the next chapter, we extend the core VTCL (designed originally for batch transformations) to event-driven transformations that define a novel specification and execution framework. Chapters 6 and 7 use this extended syntax in their illustrative examples.
Chapter

5

Event-driven model transformations by
incremental pattern matching

Contributions of the current chapter  In the current chapter, we introduce the novel concept of event-driven model transformations, which provide the basis for the rest of the model transformation-related contributions of the thesis (Progress map 2). Event-driven transformations (EDTs), in contrast to the traditional batch execution model of model transformations, run transparently in the background, and can automatically react to arbitrarily complex changes of the underlying model (Challenge 4). By our proposal, EDTs are adapted from the event-condition-action formulas of expert systems, and can be specified using an extension of the graph transformation language of ViATRA2. The efficient execution of EDTs is supported by incremental graph pattern matching (Contribution 1).
5.1 Foundations of incremental model transformations

Incrementality in software engineering can be intuitively defined as a computation strategy that aims to process new information in increments, in order to avoid re-computing old results that are still valid. This strategy promises an increase in computing efficiency, since it resources are (mostly) spent on the processing and propagation of changes rather than the recomputation of solutions of a problem. Hence, the design of incremental algorithms requires techniques to efficiently separate valid and invalid results (that are affected by change). The better an algorithm is at doing this, the less computing power is wasted in re-computation.

In model transformations, incrementality generally means to perform a mapping of an evolving model by reusing already computed (partial) results. The most common application scenario of such techniques is model synchronization, where incrementality means that a source-target mapping is performed by taking the changes of the source model, and computing a corresponding change of the target model, and applying that change only. The motivation is twofold: (i) the more efficient propagation of changes results in increased scalability, and (ii) since (target) models are not re-created, exogenous transformations (where models are manipulated outside the modeling framework through an interface) become feasible (see Section 8 for more details on such a scenario).

In this section, we demonstrate that incrementality can be defined for a wide range of application scenarios outside of model synchronization as well, such as model simulation and well-formedness constraint checking. To support such applications, we give a general taxonomy of incrementality in model transformations, that encompasses basic notions (Section 5.1.1), change computation techniques (Section 5.1.2), execution strategies (Section 5.1.3) and the usage of traceability models (Section 5.1.4).

5.1.1 Core concepts

In this section, we establish the core concepts that are necessary to understand the techniques and application/usage scenarios in detail. Our terminology is based on [HLR06] and [K. 06], which are centered around the traditional incremental model synchronization scenario (incremental change propagation along a source $\rightarrow$ target chain). In our work, we re-use these concepts but extend them where necessary.

Source incrementality Source incrementality means that a model transformation avoids unnecessary model traversal, and only reads those parts of the model which are relevant for change detection and propagation. Source incrementality is a design goal that emphasizes to minimize the query overhead, since the execution of model queries is computationally expensive – especially in a graph transformation context, where traditional, non-incremental pattern matching techniques are used (Section 4.1.2).

Target incrementality Target incrementality means that a model transformation does not regenerate any target model, but updates the models instead. In other words, the goal is to minimize the amount of model elements that have to be manipulated. Some examples are the following:

- Coarse grained target incrementality applies to the encapsulation of change propagation: traceability information (Section 5.1.4) is used to calculate the scope of the target model that is affected by a change in the source model. The model manipulation is restricted to this context only.
Fine grained target incrementality applies to the propagation of source model changes: moves are propagated to the target model, i.e. the transformation does not delete and re-add model elements in a new container, but simply moves them there; attribute value updates are mapped to value updates rather than deleting the old element and adding a new one with the new value.

Incremental model transformations are commonly aimed at updating existing target models based on changes in the source model [K. 06].

Merging strategy To achieve target incrementality, an incremental transformation approach creates change sets which are to be merged with the existing target model instance. Then, a merging strategy is used to decide on how these changes are applied. Such a merging strategy is typically executed automatically, even though model merging almost always involves non-determinism (i.e. a change could be applied in multiple ways producing multiple valid outcomes). Examples:

- A blind merge means to perform all operations without checking for conflicts (e.g. when an already existing element has to be created); the merging process usually stops and reports error feedback on the first conflict.
- A full merge with conflict avoidance means to perform all executable (non-conflicting) operations on the target model, including deletions, and report conflicts afterwards.
- A full merge with conflict resolution means that attempts are made to resolve conflicts by heuristics (e.g. rename conflicting model elements).
- Finally, a non-destructive merge means that only additive operations are executed on the target models, deletions are only propagated as marking certain elements to be deleted. Conflicting operations are worked around by e.g. renaming.

### 5.1.2 Change computation strategies

Change computation strategies distinguish between ways of achieving source incrementality. In order to efficiently calculate which source element may trigger changes, a transformation context that is persistent over several transformation runs (execution phases) has to be maintained, which stores the execution state of the model transformation system (e.g. variable values, partial matches). Depending on whether this is possible or not, there are two main approaches to incremental transformations, as illustrated in Figure 5.1.
CHAPTER 5. EVENT-DRIVEN TRANSFORMATIONS

Re-transformation Systems employing re-transformations lack the capability to maintain the transformation context over multiple execution runs, thus the entire transformation has to be re-run on the modified source models. This approach generates either new output models which must be merged with existing ones, or change sets which can be merged in-situ. As noted in [HLR06], since the transformation context is lost, a merging strategy has to be employed. This involves the computation of which model elements are involved in the change, and which elements should be left untouched by the transformation. Thus, the feasibility of this approach depends heavily on the trace information. For instance, in case of graph transformation, negative application conditions (NACs, Section 4.1.2.1) may be used to forbid the execution of a transformation rule twice on the same source element. An intelligent re-transformation based model synchronization approach has been proposed recently for ATL in [XLH+07a], which targets bidirectionality rather than incrementality.

Live transformation In contrast, live transformations maintain the transformation context continuously so that the changes to source models can be instantly mapped to changes in target models. Live transformations are persistent and go through phases of execution whenever a model change occurs. Similarly to re-transformations, the information contained in trace signatures is used in calculating the source elements that require re-transformation. However, as the execution state is available in the transformation context, this re-computation can be far more efficient.

5.1.3 Execution strategies

Execution strategies describe the overall lifecycle of transformation execution, i.e. they define the phases and timing of model query and manipulation operations. Traditionally, model transformation tools support the batch execution of transformation rules, which means that input is processed as a whole, and output is either regenerated completely, or, in more advanced approaches, updated using trace information from previous runs. However, especially in modern model-driven engineering scenarios, models are highly dynamic and they are evolving and changing continuously. Thus, to adapt transformation to such a rapid pace of change, batch transformations are extended by event-driven transformations that need not to be triggered explicitly, but react automatically whenever necessary.

Batch transformations Batch transformations are analogous to traditional batch program execution. They are characterized by (i) having a single entry point which specifies the first operation of an execution sequence, and (ii) a sequential control structure that may incorporate hard-coded structures as well as data (model) dependent parts (Section 4.1.4).

Event-driven transformations Event-driven transformations, on the other hand, are triggered by an event (rather than by explicit invocation). The source of such events can be the user interface, or the model itself. Such transformations run like daemons and go through phases of execution, as reactions to trigger events. Transactionality is an important aspect of event-driven transformation technology, as it defines a temporal framework around all model manipulation operations, which helps to precisely identify semantic model changes that can be processed by an event-driven system.

5.1.4 Traceability in incremental transformations

Traceability models (Section 2.6) play an important role in the design of incremental transformations. In general, they record correspondence information between models, and as such, they are used to connect corresponding source and target elements in a source → target scenario. As the presence
of such connections indicates that the mapping has been performed, they store information on the past and thus traceability links serve as the "memory" of the transformation process and are (part of) the transformation context (Section 5.1.2). More details on the usage of traceability models in change propagation is discussed in Section 6.4.1.

Technically, the generation and maintenance of traceability models is characterized by two important factors:

- **Internal vs. external traceability**: External traceability means that traceability models conform to an explicitly defined metamodel, and are stored in a separate model (that contains cross-references to source and target models). In contrast, internal traceability models are stored as parts of the host model (e.g. attributes, foreign keys) and do not have a separate metamodel.

- **Implicit vs. explicit traceability**: Explicit (rule-guided) traceability means that the generation of traceability model elements is parameterized by specification metamodels, which encode semantic information on how traceability models are to be instantiated. In contrast, implicit traceability means that the generation of traceability model elements is "hard-coded" in model transformations (an ad-hoc design pattern).

**Evolution history traceability models** Traceability models may also be used to encode generic information on model evolution, by annotating the elements of a host model with special auxiliary elements that correspond to model queries, manipulation operations or transformation rules that have been executed or applied on them. This technique is a conceptual extension of the plain correspondence modeling approach in the sense that additional semantic information is added to links. More details on this technique are discussed in Section 8.

## 5.2 Overview of event-driven transformations

### 5.2.1 General concepts

In this section, we propose a novel combination of event-driven execution (Section 5.1.3) with live computation techniques (Section 5.1.2) enabled by incremental graph pattern matching (Section 4.2), to specify and execute incremental model transformations in an intuitive and efficient way.

**Explicit specification** In addition to targeting the incremental execution of model synchronization transformations, our approach is intended to support a broader range of event-driven transformations (such as model execution, well-formedness constraint validation and other interactive scenarios). Therefore, we chose to give the transformation engineer precise and explicit control over all aspects of event-driven execution (with transparent support for live re-computation). In order to achieve this, our event-driven transformations (called triggers) are explicitly specified in an extended dialect of the VIATRA2 transformation language.

Our language (described in detail in Section 5.3) allows event-driven transformations to be specified intuitively, by relying on well-established concepts and re-using the concepts of graph patterns and graph transformations. This language also allows to flexibly mix traditional batch and event-driven transformation techniques by (i) allowing traditional transformations to control the life-cycle (starting and stopping) of event-driven transformations and (ii) allow event-driven transformation rules to reuse (call into) traditional rules. The language was designed to provide a balance between conciseness and control to support practical applications, with a special focus on source and target
incremental model transformations (Section 5.1.1), where the merge strategy can be specified by the transformation engineer.

Architecture The conceptual architecture of our event-driven transformation system is shown in Figure 5.2. Event-driven rules can be registered into the transformation space of the execution engine at any time. This transformation space provides support for batch as well as event-driven transformations with interoperability in both directions: (i) batch transformations may initialize and dispose event-driven rules, and (ii) event-driven rules may call (parts of) batch transformations freely. The transformation context is maintained and supported by the transformation space.

Transformations operate on a model store (Def. 4) that is supported by a transaction system that keeps track of all model manipulation operations applied to the model. Events are registered using this transaction system and can correspond to (i) changes of models in the model store, or, in broader terms, (ii) anything that influences the state of the model such as user actions and external messages.

Event-driven rules are, in turn, defined in terms of such events, conditions that define further (more refined) constraints on the events and their details, and actions that specify the reaction of the transformation system (these are essentially normal transformation operations as discussed in Section 4.1.4). The execution of such event-driven rules is facilitated by the transformation engine, which provides three core features: (1) the detection and processing of events, (2) evaluation of conditions on events, and (3) the execution of actions and applying their changes to the model in a transaction-oriented way.

As emphasized before, the general architecture in Figure 5.2 does not restrict the event concept,
but in this work, we focus on *model change events* that are defined on a very high abstraction level. The notions of event-driven transformation rules are shown in Figure 5.3. Event-driven rules are a specialization of graph transformation rules and reuse many related concepts such as *graph patterns* and *transformation rule bodies*.

**Events** can be defined in three major ways:

- *Registered events* are explicit symbolic identifiers that refer to a-priori known (atomic) events such as user actions or messages sent to and event-driven system. They are the most generic way of defining events, since - apart from the (symbolic) identifier - no in-depth information is modeled about the event.

- *Match set change events* are defined in terms of changes in the *match set* of graph patterns (Def. 22). Such match set changes correspond to *compound changes* (in the terminology of [Le08]), since they represent changes in the result set of arbitrarily complex (or very general) queries. In the current section, we focus on this type of event definition.

- *Modeled events* are model-based representations of the above two. The crucial difference is that they do not refer to temporally observable events of the future, but to already registered (and serialized) events of the past. This notion will be used in Chapter 8.

**Definition 25** (Model change event definition) A model change event definition $E$ is a tuple $E = (GP, Dir)$, where $GP$ is a graph pattern (Def. 21), and $Dir$ represents the direction of change:

- a *rise* $\in$ $Dir$ event corresponds to a new match in the match set,

- a *fall* $\in$ $Dir$ event corresponds to a loss of a previously existing match.
Conditions  focus on the details of model change events, to allow the transformation designer to impose finer-grained restrictions on those elementary model manipulation operations that contributed to the observation of the event. An operation condition refers to a model element variable (Def. 21) of the graph pattern of the event definition, and describes an elementary operation type (kind) that is to be located in the sequence of operations of the transaction that generated the event.

**Definition 26 (Condition definition)** A condition definition is a tuple \( CD = (Var, OpKind) \), where \( Var \) is a model element variable of a graph pattern \( GP \), and \( OpKind \) represents the type of the elementary change over which the condition is defined:

- a \( CREATE \in OpKind \) condition means that the model element bound to \( Var \) has been newly created;
- a \( DELETE \in OpKind \) condition means that the previously existing model element bound to \( Var \) has been deleted;
- a \( UPDATE \in OpKind \) condition means that the VPM name or value (Def. 12) of the model element bound to \( Var \) has been changed.

**Definition 27 (Event-driven graph transformation rule)** An event-driven graph transformation rule \( EDR \) is a tuple \( EDR = (GT, ED, \{CD\}) \) that extends the definition of a graph transformation rule \( GT \) (Def. 23) with the followings: (i) a precondition event definition \( ED \); (ii) a set of operational condition definitions \( \{CD\} \). The actions of the rule are exactly those of the graph transformation rule \( GT \).

### 5.2.2 Execution of event-driven rules

#### 5.2.2.1 Transaction management

In order to be able to perceive changes in the match set of a pattern over a complex model manipulation operation, such as the execution of a graph transformation rule or a complex editing operation, the model management system has to support transactions. A transaction is defined as an ordered sequence of atomic model manipulation operations (e.g. create node, edge, instance-type-supertype relation, update attribute, etc.), followed by a commit command. On the implementation level, the model management framework (in our case, VIATRA2) has to ensure that all model manipulation occurs within a transaction.

**Definition 28 (Elementary operation)** An elementary operation \( Op = (OpKind, me) \) is an atomic model manipulation operation in a modelspace \( MS \) that affects a single model element \( me \in ME \).

**Definition 29 (Transaction)** A transaction \( T \) is an ordered series of elementary operations that describe a modification trajectory that transforms the initial state of the model space \( MS_{pre} \) (registered at the beginning of the transaction) to the final state \( MS_{post} \) (registered at the end, or commit point of the transaction). Formally: \( T = (MS_{pre}, MS_{post}, ops) \), and it must hold that all model elements referenced by elementary operations are valid in either the pre-state, the post-state, or both: \( \forall me : \exists op \in ops, op.me = me \Rightarrow me \in MS_{pre}.ME \cup me \in MS_{post}.ME \).

The operational workflow of the event-driven transformation system is shown in Figure 5.4. The model store provides synchronous information on model manipulation transactions to the event-driven execution system through notifications (Section 5.2.3). Using this facility, transactions can be processed in detail, with every elementary operation visible.
5.2. OVERVIEW OF EVENT-DRIVEN TRANSFORMATIONS

5.2.2 Detection of model change events by delta monitors

After a transaction has reached its commit point, the system evaluates the changes in the match sets of precondition patterns of triggers registered in the trigger queue. The evaluation is linear, i.e., more registered triggers increase the execution time linearly (maybe also sublinearly, if RETE networks belonging to different patterns are interleaved by sub-pattern reuse). Since the RETE networks are updated after each atomic model manipulation operation, a match set may experience transient changes while a long transaction is running. By our approach, only the effective changes are considered; thus, even if a new match is generated while a transaction is running, if that match is subsequently lost, the system will not process it for triggers. This mechanism is provided by the matching set delta monitor (on the left in Figure 5.4), which computes the net changes that occurred during a transaction.

**Definition 30 (Model change event occurrence)** A model change event occurrence of a model change event definition $ED$ is observed if the elementary operations of the transaction together result in a newly found match ($rise$ events) or the loss of a previously existing match ($fall$ events) in the match set (Def. 22) of the precondition pattern $GP$ of the event definition. Formally: along a given match $m$ of the precondition graph pattern $GP$, an event occurrence holds over a transaction if:

$$T|^mED \iff \begin{align*}
\text{rise} : & \quad MS_{\text{pre}}^m \notmodels GP \land MS_{\text{post}}^m \models GP \quad (\text{new match found})
\end{align*}$$
In general, a change event occurrence holds if there is at least one match $m$ for which the above conditions hold. Formally, $T|\models ED \iff \exists m : T|\models ED$.

5.2.2.3 Evaluation of operational conditions

Operational conditions are evaluated by analyzing the sequence of operations that constitute the transaction over which the matching set changes have been observed (Transaction$_1$ in Figure 5.4). This analysis is done by a linear traversal along the chain of elementary operations of the transaction.

- **create(ME)** conditions are fulfilled if there is an elementary operation which corresponds to the creation of the model element $ME$;
- **delete(ME)** conditions are fulfilled if there is an elementary operation which corresponds to the deletion of the model element $ME$;
- **update(ME)** conditions are fulfilled if there is an elementary operation which corresponds to the update of the name or value of the model element $ME$.

**Definition 31 (Condition occurrence)** A condition occurrence holds along a given match $m$ over a transaction $T$ if there is an elementary operation $op$ in $T$ that satisfies the condition definition $CD$. Precisely, the referenced model element of $op$ is bound in $m$ to the variable $Var$ referenced by $CD$, and the operation kinds are equal. Formally, $T|\models CD \iff \exists op \in \mathcal{ops} : m(CD.Var) = op.ME \land CD.OpKind = op.OpKind$.

5.2.2.4 Execution modes and ordering

After the changes have been evaluated, the execution engine processes triggers registered in the system and selects those with a precondition activated by the processed matching set changes, and prepares them for execution based on the execution mode. The following output is computed:

- all those triggers which have been activated are prepared for execution;
- for each activated trigger, the set of activated pattern matches $\{m\}$ is also computed (by querying the production nodes of the RETE network).

Action sequences of activated triggers can be executed in three modes (Figure 5.5). In the depicted scenario, we assume that there are three active triggers (T1–3) with their action sequences (AS1–3 respectively). After a transaction, the system encounters a new match (M(T1)–M(T3)) for each of the three triggers.

**Ordering** In **serial ordering** (Figure 5.5a), the action sequences are executed in separated transactions according to the priority order. After each commit point, the system re-evaluates all trigger conditions. In this ordering, conflicts between competing triggers are eliminated (since the checks may reveal, for instance, that M2 was invalidated while AS1 was executed). However, a circular activation of triggers may result in infinite loops in case of serial execution ordering.

In contrast, **pseudo-parallel ordering** (Figure 5.5b) means that action sequences are executed in a single transaction with a common commit point. In this case, conflicts may occur, and they need to
be accounted for by the transformation designer. On the other hand, the execution is faster than in serial ordering, since no intermediate checks are performed.

**Execution modes** Race conditions may also arise for multiple matches for a single trigger. In Figures 5.5c and 5.5d, trigger T1 has been activated for matches \(M_1 - M_3\).

- In **choose mode** (which is a special case of *iterate mode* and not depicted in Figure 5.5), a randomly selected match is used and its action sequence is executed in a single transaction. All other matches are ignored.

- In contrast, **iterate mode** (Figure 5.5c) implies that if the rest of the matches are not invalidated, their respective actions are also executed one by one in separate transactions.

- Finally, in **forall mode** (Figure 5.5d), the execution of all action sequence and match combinations occurs in a *single transaction* without regard for match invalidations and thus the possibility of conflicts which may cause a run-time error.

### 5.2.2.5 Execution algorithm

The complete execution scheme for the event-driven transformation system is shown in Figure 5.6. The operation of the transformation space is defined by a *queue automaton with guards* [FE81], where the a *last-in-first-out queue* is used to store activated *trigger instances* (a trigger instance corresponds to a trigger-graph pattern match pair). In Figure 5.6, the following notation is used:

- **States** are shown by blue rectangles. *Initialization states* and *finalization states* for triggers are shown as white and blue circles, respectively.

- **Control flow transitions** are shown by blue arrows, with *guards* shown in brackets. The single red arrow corresponds to an *instantaneous* transition.
• **Decision states** are shown by blue rhomboids, **decision conditions** are printed in italics. **Decision outcomes** are printed as normal labels (Y/N).

• **External data stores** are shown by grey cylinders.

• **The trigger queue** (a special data store) is shown as a red cylinder.

• **Data flow connections** between data stores and states are shown as grey dashed arrows.

The operation of the event-driven execution engine goes through three main phases: (shown on the left in Figure 5.6).

1. **Transaction processing**  The initial and default state is the **IDLE** state. Whenever a new transaction is started in the model space (guarded by \([Tbegin()]\)), the event-driven engine will begin registering the changes in the **T running** state. Once the transaction has been committed (\([Tcommit()]\)), the processing and evaluation of event conditions and operational conditions will commence (in the **EVAL** state) – if the transaction has been aborted, the system returns to **IDLE** (\([Tabort()]\)).

2. **Event and condition evaluation**  The **EVAL** state represents the computations required for the evaluation of pattern match set changes (using the **RETE engine** as a data store) and operational conditions (using the **transaction log** as a data store). If there is an activated trigger instance (i.e.
5.2. OVERVIEW OF EVENT-DRIVEN TRANSFORMATIONS

either a rise or fall has been registered for one of the triggers registered in the system through the trigger registry, the system will proceed to trigger instance scheduling – if no such triggers are detected, the system returns to the IDLE state.

3. Scheduling and execution In the PUSH Queue state, the system pushes activated trigger instances (trigger-new/lost match pairs) to the trigger queue according to the execution mode (Section 5.2.2.4). After the triggers have been scheduled, the system returns to the IDLE state.

In the IDLE state, an instantaneous guard (red arrow) represents a transition to the Trigger queue empty decision state. This transition is fired instantaneously whenever the trigger queue is manipulated with mutual exclusion to the \[T\text{begin}()\] guard, which means that (i) the automaton will loop along the trigger queue processing path as long as the queue is non-empty, but (ii) after a transaction has begun, the trigger queue processing is suspended to ensure that there are no concurrent transactions in the system.

The trigger processing loop starts with the POP Queue state where a trigger instance is removed from the queue and its action part is executed according to ordering rules (Section 5.2.2.4). Trigger action sequences are also executed in transactions (represented by the TR.A: T begin() transition in Figure 5.6).

5.2.3 Implementation details

Observing model change events In order to increase the reusability of event-driven execution techniques, a straightforward approach is to make use of notification mechanisms found in many model persistence frameworks (e.g. EMF Notification \[?\]) that provide call-back interfaces (delegates) on model manipulation operations, or transactions. By this technique, the source of events is the model itself (rather than a custom mechanism tied to the user interface or other external components), which means that the transformation execution only depends on (changes in) the contents of the model. This provides a much more flexible approach for the integration of such transformations.

Detecting changes in match sets Changes in the match set can be tracked using the RETE network. A model change occurs if the match set is expanded by a new match or a previously existing match is lost. Since a graph pattern may contain multiple elements, a change affecting any one of them may result in a change in the match set. The RETE-based incremental pattern matcher keeps track of every constraint prescribed by a pattern, thus it is possible to determine the set of constraints causing a change in the match set.

The operation of the RETE-based pattern matcher also solely depends on the notification mechanism. If such events can be observed, the updates are automatically propagated along the RETE network (independently of transaction boundaries), and the contents of production nodes are guaranteed at any given time to contain the valid match set of the graph pattern.

The RETE-based pattern matcher was primarily developer for use with the ViATRA2 framework. However, it has been implemented in a way so that core algorithms were kept isolated from ViATRA2-specific details, and this enabled an adaptation to the EMF API \[11\].

Evaluation of operational conditions Operational conditions can be evaluated using two common mechanisms available in model management frameworks such as EMF or even relational database systems:
• As mentioned before, a (transaction-oriented) notification API provides synchronous feedback on elementary model manipulation operations using delegates or call-back methods. These notification objects can be collected by listeners for further analysis when a transaction commit point has been reached.

• In interactive environments (such as EMF-based editors), a command stack API may also be available, which provides services for listening to commands (objects that represent user-originated operations) that get applied to the model. Such command objects can be directly queried for elementary operation analysis.

The VIATRA2 framework provides high level support for both techniques, and the prototype implementation relies mostly on the notification API. However, the concepts and techniques outlined in this work can be adapted to other modeling environments where such facilities are available.

Live recomputation Live transformation execution requires the continuous maintenance of the execution context to avoid the necessity of model merging in target models. In our approach, this context contains:

• global variables, which are persisted to enable the transformation engine to store (global) cached values.

• pattern variables, which are maintained by the incremental pattern matching engine after each atomic model manipulation operation. This means that the matches stored in a given pattern variable are always updated and the match set of any pattern can be retrieved in constant time.

As a result, the computation required to initialize and execute the incremental transformation sequence after a change is fast, since pattern matching, the most cost-intensive phase of the transformation, is executed in linear time with respect to the size of the match set.

5.3 Graph triggers: an event-driven transformation language

In this section, we introduce an extension to the VIATRA2 Textual Command Language (Section 4.1.1) to specify event-driven transformations. In our approach, the basic functional unit is the trigger. It is a specialized variant of the graph transformation rule, with a different interpretation.

In triggers, the precondition pattern is interpreted (together with a combination of annotation modifiers) as the specification of events, and the action part (augmented with when clauses that define operational conditions) specifies actions (sequences of VIATRA2 transformation steps, including simple model manipulations as well as the invocation of complex transformations).

A simple trigger example In the simplest case, triggers behave like normal graph transformation rules in an automated execution mode, and are invoked whenever a change in the match set of their precondition pattern occurs. A simple example of such an execution style is shown in Figure 5.7. In this example, we rewrote the sample Petri net simulator of Figure 4.13 to an event-driven simulator: whenever a new fireable transition is detected in the model (signalled by a new match of the fireable(Tr) pattern of Figure 4.2), it is automatically fired (by a call to the firetransition(Tr) rule of Figure 4.13). If multiple transitions become enabled, the execution will randomly choose one of them. This event-driven transformation will terminate only if there are no fireable transitions left.

Any graph transformation rule written in VTCL can be converted to an event-driven rule by adding the @Trigger annotation to the declaration.
5.3. GRAPH TRIGGERS: AN EVENT-DRIVEN TRANSFORMATION LANGUAGE

@Trigger (sensitivity = rise, execution = choose)
grule fireAsLongAsPossible () =
{
  precondition find fireable (Tr)
  action {
    call firetransition (Tr);
  }
}

Figure 5.7: Simple trigger example

Annotation modifiers  @Trigger annotations accept the following parameters (key-value pairs) that are written in a Java 5-like syntax. These modifiers affect the semantics and thus the behaviour of event-driven graph transformation rules.

- **Sensitivity** (rise | fall | both): Rise triggers are activated whenever a new match (of the precondition pattern) is encountered; fall triggers are activated when a previously existing match is lost; both triggers activate on rises and falls as well.

- **Priority** (integer): Defines a precedence relation on multiple active triggers. From those triggers that are activated at the same time, the one with the highest priority value will run first.

- **Mode** (always | once): Defines whether a trigger is continuously scheduled for execution, or it is executed only once and then it becomes disabled forever. By default, all triggers are assigned to run in always mode.

- **Execution** (choose | iterate | forall): the distinction between these execution modes corresponds to the various invocation modes of traditional graph transformation rules discussed in Section 4.1.4.3. In the choose mode, a single (randomly picked) match (among the newly found, or lost matches) will be picked for execution, in iterate mode, all such matches will be processed one after another, while in forall mode, this processing will be done a pseudo-parallel manner (see Section 5.2.2.4 for more details).

5.3.1 Execution context

The system tracks changes in the match sets of patterns and executes the action sequences in a persistently maintained execution context. This context consists of pattern variables (continuously maintained by the RETE network) and persistent variables (ASM functions Section 4.1.4.1). These can be used to persist information between transformation execution sequences (transactions) and thus carry traceability information along the execution path.

Since pattern variables (Tr in Figure 5.7) are also part of the maintained context, their values are instantly available for processing during the execution of an event-drive rule. Hence, the execution of such rules is incremental, since the underlying RETE-based pattern matcher maintains the matches for the precondition pattern (fireable) incrementally and no model traversal is necessary for the computation of events and the evaluation of conditions.

Using ASM functions in the execution context  An example usage of ASM functions in the execution context is shown in Figure 5.8. We extended the previous example of Figure 5.7 to include a simple termination conditions where each transition is only allowed to fire at most once. In this
example, we use the \texttt{fired/1} ASM function to store already fired transitions to ensure that no Transition will be fired more than once. In the \textit{action} part, an if-check is added to query (using \texttt{fired/1}) whether the fireable transition \textit{Tr} has been fired before, and the call to the \texttt{firetransition()} rule is extended with an update of the corresponding slot of \texttt{fired/1}.

\begin{verbatim}
// store whether a transition has been already fired
// Transition -> Boolean
assfunction fired/1;

@Trigger(sensitivity=rise, execution=choose)
grule fireAsLongAsPossible() =
{
  precondition find fireable(Tr)
  action {
    if (fired(Tr)==false) seq {
      call firetransition(Tr);
      update fired(Tr) = true;
    }
  }
}
\end{verbatim}

Figure 5.8: Triggers and ASM functions

The contents of \texttt{fired/1} are preserved in-between executions of this rule, which ensures that the execution will terminate after all transitions that have become fireable at some point have been fired once.

5.3.2 Complex change detection

To detect complex model changes, the transformation developer can make use of the \textit{rise} and \textit{fall} triggers and \textit{when} clauses, that define operational conditions on the sequence of elementary model changes that constitute the \textit{net change} corresponding to the newly found (or lost) match.

5.3.2.1 Creation

In order to detect that a model element (or a certain configuration of multiple model elements) has been added to the model space, a \textit{rise} sensitive trigger is defined with the corresponding precondition pattern. As a pattern may refer to multiple model elements, \textit{when(create()) clauses} (used in the \textit{action} part) allow the transformation designer to impose creation conditions on each model element (identified by a pattern variable) to distinguish between various model manipulation operation sequences that caused the activation of the trigger.

An example of this technique is illustrated in Figure 5.9. Here, we use the \texttt{PetriNetPlace(PN,P)} pattern as the precondition to define a simple event-driven transformation rule that prints a log message whenever a new Place is added to the model. The trigger will be activated whenever a new match of the \texttt{PetriNetPlace} pattern is observed; however, as such a new match can be the result of many different elementary operation sequences (e.g. if a previously existing Place is assigned to a new Net), we impose a further condition to only print the log message if the atomic event \texttt{create(P)} (corresponding to the creation of a new model element that is mapped to the pattern variable \textit{P}) was indeed part of the sequence that caused the activation of the trigger.
5.3. GRAPH TRIGGERS: AN EVENT-DRIVEN TRANSFORMATION LANGUAGE

5.3.2.2 Deletions

Analogously to the technique presented for creations, the deletion of model elements (or disappearance of model element configurations) can be used as activation conditions of fall sensitive triggers.

```plaintext
@Trigger(sensitivity=fall, execution=iterate)
gtrule lostToken() =
{
  precondition find PetriNetPlace(PN,P)
  action {
    when(delete(P)) seq {
      log("A place \"+P+\" has been deleted, from Petri net \"+PN\";)
    }
  }
}
```

Figure 5.10: Trigger deletion

Figure 5.10 illustrates an example of this technique. In this case, we defined an alternative version of the example of Figure 5.9: this trigger will fire after a previously existing match of PetriNetPlace has been lost, and this loss was due to the deletion of a Place (expressed by the `when(delete(P))` condition).

5.3.2.3 Attribute updates

The language also supports the event-driven detection of attribute value changes (in VIATRA2, attribute values are stored by the `value` VPM core function, see Def. 12). Again, the key technique is the usage of the `when(update())` clause, which defines a condition that corresponds to an update operation on a model element (name or value change) identified by a pattern variable.

In Figure 5.11, we reuse the capacity constraint example in Figure 4.4 of Section 4.1.2.3 to define a simple event-driven rule similar to the previous introductory examples. The `when(update(CC))` clause is used to identify those cases where the change (of both sensitivity) the matching set of the CapacityConstraint pattern was caused by a change of the CapacityConstraint element. In such cases, the `:old` and `:new` discriminators may be used to refer to the values of the `value` VPM function.
5.3.3 Trigger lifecycle management

Triggers, as shown in Figure 5.6, have a simple life-cycle consisting of two states (ACTIVE and INACTIVE). This lifecycle can be managed using VTCL using native rules (Section 4.1.4.4) and queried using a native function as follows:

- **Controlling states** can be achieved by two native rules:
  - A trigger can be started (moved from INACTIVE to ACTIVE state) by passing its identifier to the startTrigger native ASM rule, any time. The state transition will occur after the execution has reached the IDLE state.
  - A trigger can be stopped (moved from ACTIVE to INACTIVE state) by passing its identifier to the stopTrigger native ASM rule, any time. The state transition will occur after the execution has reached the IDLE state.

- **Querying the current state of a trigger:** the getTriggerState native function can be used for this purpose, which returns a symbolic constant ACTIVE or INACTIVE depending on the actual state of the trigger.

The usage of the language facilities above is illustrated in a small example in Figure 5.12.

5.4 Performance evaluation

5.4.1 Description of the benchmarking scenario

The Object-to-Relational schema mapping (ORM) benchmark, as presented here, is an extension of a graph transformation benchmark proposed in Sec.4 of [VSV05a]. The original transformation processed UML class diagrams to produce corresponding relational database schemas, according to the known mapping rules. Since a straightforward application of the event-driven transformation approach is the synchronization between source and target models, we extended the benchmark by...
5.4. PERFORMANCE EVALUATION

two additional sequences: (i) after the initial mappings are created, the source models are modified, and, in an additional pass, (ii) the system has to synchronize the changes to the target model (i.e. find the changes in the source and alter the target accordingly). An incremental approach can perform this efficiently as it can track changes in the source model so that the model parts affected are instantly available for the synchronization sequence.

Test case generation In order to produce sufficiently large model graphs for the measurements, we implemented a simple generator as described in [VSV05a]. By this approach, a fully connected graph is created, i.e. for $N$ UML classes, $N(N - 1)$ directed associations are defined (with each association represented as three nodes – an association node and two endpoints). Additionally, each UML class can reference $K$ attributes, thus, for a given $N$ and $K$, $N + 3N(N - 1) + NK$ nodes and $4N(N - 1) + NK$ edges are created (Figure 5.13). Although the model produced is not realistic in the sense that very few practical UML class diagrams are fully connected, the method is quite efficient in creating large graphs quickly.
CHAPTER 5. EVENT-DRIVEN TRANSFORMATIONS

Execution phases  The transformation sequence is comprised of four main phases:

1. The generation phase creates the model graph.

2. The build phase creates the initial mapping of the UML model into the relational schema domain, with reference models connecting mapped model objects.

3. The modification phase modifies the UML models programmatically to emulate user editing actions.

4. Finally, the synchronization phase locates the affected model elements and makes changes in the schema model accordingly.

Characteristics  For this benchmark, we compare the execution times for the last (synchronization) phase. In order to scale the synchronization sequence as the model size grows, we designed the modification sequence to extend roughly linearly with the model. Thus, in the default case, it is composed of the following operations: (i) first, one third of generated classes, along with their attributes and referenced associations are deleted; (ii) then, one fifth of remaining associations are deleted; (iii) next, every second attribute is renamed; (iv) finally, a new class is added and a new fully connected graph is created (with the remaining UML classes and the newly added class as nodes, ignoring existing associations).

In order to give a comparable description of our benchmarks with the standard ones defined in [VSV05a], we also use feature matrices to describe the characteristics of these test sets. The definition of the features are the following:

- Pattern size, or the number of nodes and edges in the LHS graph, is a critical factor in the runtime phase of pattern matching.

- The maximum degree of nodes (fan-out) in the model is the number of edges that are adjacent to a certain node.

- The third feature is the number of matches during the test case execution.

- The length of the transformation sequence also affects the overall execution time. For example, with large number of rule applications, the relative cost of one-time overhead of the pattern matcher is decreased.

The feature matrix is shown in Figure 5.14. Note that if the characteristics of a feature depends on the concrete parameter settings of the test case, then it is called parameter dependent (marked PD).

Transformation rules for synchronization  In incremental synchronization, to avoid rebuilding target models in each pass, a traceability reference model is used to establish a mapping relationship between source and corresponding target model elements (Figure 5.15). With correspondence edges, it is possible to track changes in both the source and target models: for instance, the graph pattern in Figure 5.16 matches tables in the schema model which are no longer referenced by classes or associations in the UML models (orphan tables).

Similarly, a newly created class may be matched by a negative condition forbidding the existence of a mapped table. Renames (value changes) may be expressed e.g. by matching for both the attribute
5.4. PERFORMANCE EVALUATION

<table>
<thead>
<tr>
<th>Paradigm Features</th>
<th>ORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHS size</td>
<td>large</td>
</tr>
<tr>
<td>fan-out</td>
<td>medium</td>
</tr>
<tr>
<td>matchings</td>
<td>PD</td>
</tr>
<tr>
<td>transformation sequence length</td>
<td>PD</td>
</tr>
</tbody>
</table>

![Figure 5.14: Feature matrix for the ORM benchmark](image1)

The ORM synchronization benchmark was executed with model sizes up to 67800 nodes (with edges, the total model size is 157800 model elements) that were first generated (Figure 5.17), and the execution time for the build and synchronization phases was measured.

In order to emphasize the performance improvements of incremental pattern matching in general, and the event-driven transformation engine (built on the RETE implementation) in particular, we conducted the same measurements using the default, local search-based pattern matcher implementation in VIATRA2 in batch execution mode (VIATRA/LS), and using the RETE-based event-driven engine (VIATRA/RETE).

All measurements presented in the followings have been carried out on Mac OSX, Java SE 1.6.0 running on an Inter Core2 Duo CPU running at 2.4 GHz, with 4GBs of RAM. The results are shown in Figure 5.18 (model size is the total number of nodes).

It is revealed that the scaling characteristic of both phases is exponential for VIATRA/LS and linear for VIATRA/RETE. With respect to synchronization, the constant difference between the build
and sync phases for VIATRA/RETE means a constant multiplier; thus, since the model elements affected by the modification sequence are a linear fraction of the whole model, it can be concluded that the execution time for the synchronization process is a linear function of the model elements affected (as expected), and independent of the size of the rest of the model.

VIATRA/LS, on the other hand, exhibits an ever increasing time difference between build and sync; thus, the time taken for the synchronization process increases exponentially with the number of affected model elements (as expected, since in the case of local search, the system has to locate the changed elements first, which is an additional graph traversal). It is important to note that for small to medium model sizes (which, in many practical scenarios when the models are mainly edited by humans, may be considered to be below the 5000 node count range), the event-driven VIATRA/RETE engine can perform a synchronization affecting a considerable portion of the model in the 10-500 msec range which makes the approach very suitable for interactive applications.

In addition to execution times, the memory consumed by the Java Virtual Machine was also recorded. The sequence for the RETE matcher (75, 100, 114, 245, 490, 750, 1000 megabytes respectively for model sizes from 85 to 67800 nodes) shows a linearly expanding RETE network as the node count
grows, which is in-line with our expectations based on the nature of the RETE building algorithm (note that the above figures include the whole user interface with a complete Eclipse instance).

5.5 Related work

To process model change events, the simplest approach is to use explicit events that are signalled using some software mechanism, e.g. function calls, message passing or inter-process signalling. As modeling environments are typically event-driven graphical user interface applications, events can be derived from editing operations by registering programmatic "hooks" (listeners) to appropriate interfaces provided by the API.

In fact, as many model transformation frameworks (e.g. TIGER [EEHT05]) allow the direct execution of transformation rules from the UI (without explicit control structure), this can be considered a primitive form of event-driven transformations. However, a more sophisticated and widely known extension of this approach is deLara’s work on event-driven grammars [GdL04], where (i) events are registered through an adaptor interface and then persisted as special elements in the model; and (ii) the execution of transformation rules (or rule hierarchies connected by control structure) are triggered by a specially augmented model editing user interface (which also serves as the "source of events").

However, the significant limitation of these approaches is that they require extensive customization outside the modeling and model transformation domain (i.e. adaptor interfaces to the transformation execution engine and also GUI hooks). This drastically reduces the applicability of the approach in scenarios where there is limited programmatic access to the model transformation engine or the "source of events". Moreover, the specification formalism for event-driven rules itself is specific to the execution environment and is difficult to be re-used.

Event-driven techniques, which are the technological basis of live model transformations, have already been used in many other fields of computer engineering. In relational database management systems (RDBMS), even the concept of triggers [GMUW01] can be considered as simple operations whose execution is initiated by events. Later, event-condition-action (ECA) rules [DGG95, ABB06] were introduced for active database systems as a generalization of triggers, and the same idea was adopted in rule engines [SB05] as well. Specification of live model transformations is structurally and conceptually similar to ECA rules as discussed in Section 5.2. However, ECA-based approaches lack the support for triggering by complex graph patterns, which is an essential scenario in model-driven development.

In case of live transformations, changes of the source model are categorized as (i) an atomic model update consisting of an operation (e.g. create, delete, update) and operands (model elements); or, more generally, (ii) a complex sequence (set, transaction) of such atomic operations. To execute an incremental update, an atomic or complex model change has to be captured and processed. For this purpose, the following approaches have been proposed in case of declarative transformation languages:

The PROGRES [Sch90] graph transformation tool supports incremental attribute updates to invalidate partial matches in case of node deletion immediately. On the other hand, new partial matches are only lazily computed.

The incremental model synchronization approach presented in [GW06] relies on various heuristics of the correspondence structure interconnecting the source and target models using triple graph grammars [Sch94]. Dependencies between correspondence nodes are stored explicitly, which drives the incremental engine to undo an applied transformation rule in case of inconsistencies. Other
In relational databases, materialized views, which explicitly store their content on the disk, can be updated by incremental techniques like Counting and DRed algorithms [GMS93]. As reported in [VV04b], these incremental techniques are also applicable for views that have been defined for graph pattern matching by the database queries of [VFV05]. The use of non-materialized views have been discussed in [JJS06].

In [GdL07], user-guided manipulation events are directly represented as model elements in the model store, while triple graph grammars [Sch94] are extended to event-driven grammars to determine the kind of event and the model elements affected. Change detection is directly linked to user interface events as this approach primarily targets (domain-specific) modeling environments. Note that this approach, does not rely on live transformations since the transformation context is not preserved; instead, the underlying ATOM3 [dLV02] engine is started whenever an event from the UI is received. The idea, however, could be used in a live transformation environment.

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. Although it is not a live transformation approach, it could benefit from being implemented as such.

[HLR06] proposes a more general solution where fact addition and fact removal constitute an elementary change. Since the underlying TefKat [Thej] tool uses a transformation engine based on SLD resolution, a fact change may represent atomic updates (involving a single operation) as well as more complex changes, since a fact may encode information about multiple model elements (such as a complex pattern describing a UML class with attributes). This approach is only applicable to fully declarative transformation languages, since incremental updates involve the processing and modification of the SLD resolution tree (which, in broad terms, can be thought of as a special structure storing the whole transformation context).

[Egy06] describes a special application of incremental updates for the consistency checking of UML models. The approach provides a rule-based formalism to specify well-formedness constraints which are evaluated instantly after model modifications. Our demonstrating example illustrates how specialised transformations can be applied to a similar problem, but on a higher abstraction level.

**Graph transformation benchmarking** Some of the measurements in the current chapter are conceptual continuations of the comprehensive graph transformation benchmark proposed 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 [GBG+06]. A similar approach to graph transformation benchmarking was used for the AGTIVE Tool Contest [The07], including a simulation problem for the Ludo table game. Our Petri net firing test case is better suited for benchmarking performance since it can be parameterized to scale up to large model sizes and long transformation sequences.

**Event-driven transformations outside of VIATRA2** The core concepts laid out in this chapter can be adapted to facilitate event-driven execution in other model transformation tools as well. The key of the adaptation is to separate the transformation into two distinct phases: model queries and model
manipulation sequences. Model queries need to be implemented by an incremental algorithm (such as the adaptation of our RETE-based pattern matcher for EMF: EMF-IncQuery [11], a similar facility of the GROOVE tool [Ren04a], or even the RETE implementation in the JBoss Drools engine [DRL10] could be used for this purpose). Model manipulation sequences can be triggered by listening to high-level changes in the query result sets. These manipulation sequences need to be directly invokable through an API, which is the case for many EMF-based model transformation tools (e.g. ATL, Epsilon, and others).

**Summarizing comparison with related work** The main novelty of the current approach to related work is as follows: (i) our approach defines a general notion of event, based on the changes in the match set of a graph pattern (declarative model query) – this represents a significant generalization over previous work based on registering only elementary manipulation actions or action sequences; (ii) the approach combines the declarative event specification with further filtering options (conditions) that allow a more concise definition by distinguishing between transactions resulting in an identical net change; (iii) compared to expert systems, our approach re-uses declarative specifications even for the modification, from graph transformation languages.

### 5.6 Summary

In this chapter, we described a novel framework for event-driven model transformations supported by incremental pattern matching. Our approach provides high level supports for the formal specification of event-driven transformation systems in a flexible way: compound changes can be easily defined as preconditions by graph pattern model queries, augmented by elementary changes (operational conditions) that help to simplify by reducing the reliance on traceability models. The language gives precise control to the transformation designer, and is built on previous concepts, integrated into the framework of the VIATRA2 transformation language, and it supports the re-use of and interaction with traditional transformations. The execution infrastructure is backed by a high-performance incremental pattern matching engine that supports scalable and live execution [21,17,6].

The results of this chapter are targeted towards Challenge 4 and can be formulated as thesis contributions as follows:

1. I defined the **concepts for event-driven graph transformations based on incremental pattern matching** [20]. By this approach, changes of the model graph are detected by changing graph pattern match sets. Model transformation actions can make use of the delta of the match set, i.e. the exact specification of the model elements affected and the kind of change (Section 5.2).

2. I defined **execution semantics for event-driven graph transformations** [7], based on event-condition-action formulas and a queue automaton model (Section 5.2.2).

3. I elaborated a **graph transformation-based language for event-driven transformations** [20,7,4], by adapting event-condition-action formulas. This specification language is based on the model transformation language of the VIATRA2 framework, and provides a formal approach for defining graph triggers as the basic units of event-driven live graph transformations (Section 5.3).

4. Based on the transformation language and its execution semantics, I proposed **efficient implementation techniques and a software framework for event-driven graph transformations** (Section 5.2.3). I verified the efficiency of the implementation using benchmarking techniques (Section 5.4 [21,6]). The prototype implementation is part of the VIATRA2 framework [17,7].
Successive collaborative research ([4,2]) has extensively used these foundations. The implementation of the event-driven transformation framework was carried out jointly with András Ökrös, who was an MSc student under my supervision. The incremental graph pattern matcher is part of the PhD work of Gábor Bergmann.
Chapter 6

Synchronization between abstract and concrete syntax

**Progress map 3**: Separation of abstract and concrete syntax representations

**Contributions of the current chapter** The current chapter introduces an application of event-driven transformations to a model synchronization problem that addresses an important language engineering challenge, the *complete separation of abstract and concrete syntax representations* (Progress map 3, Challenge 2). EDTs are complemented with a metamodel-based parameterization library that specifies how traceability links should be created and maintained, allowing arbitrary synchronization semantics to be implemented (Contribution 4). By this approach, abstract and (graphical) concrete syntaxes of a DSML can be handled independently, but still coherently, in an interactive editing environment (ViatraDSM).
6.1 Introduction

6.1.1 Motivation

Domain-specific language engineering frameworks provide refined techniques for developing new languages based on the clear separation of conceptual elements of the language called abstract syntax (or conceptual model) and their visual representation called concrete syntax (or diagram model). This separation is achieved by the precise management of traceability information between the abstract and concrete syntax using so-called mapping models (or trace models).

Mapping models in the context of DSMLs can also be categorized from traditional traceability perspectives. For instance, first Eclipse-based DSMLs based upon the Graphical Editing Framework (GEF) used internal traceability, when concrete syntax elements (Java objects) had a direct reference to abstract syntax elements. More recent approaches like the Graphical Modeling Framework have opted for external traceability when the elements of the abstract and concrete syntax are interrelated via a separate mapping model.

However, unfortunately, even state-of-the-art DSM frameworks such as GMF impose severe restrictions on traceability links between elements of the abstract syntax and the diagram models. For instance, each concrete syntax element in a diagram must correspond to exactly one element in the underlying abstract syntax. Moreover, industrial DSM frameworks mostly provide automated synchronization between the abstract and concrete syntax in one direction: (i) changes of the diagram model initiated from the editor are propagated to the underlying abstract syntax, but (ii) direct changes of the abstract syntax frequently corrupt domain-specific editors (direct changes to the abstract syntax are usually due to editing actions in other concrete syntax representations, such as another diagram or a textual view, or may be results of automated model transformations). As a result, the use of DSM frameworks for developing complex, industrial strength visual modeling languages requires significant expertise and additional manual programming effort to overcome such difficulties.

The first claim for the current chapter is that most of these problems are caused by the very simplistic handling of mapping models in DSM frameworks. For this purpose, we propose a mapping metamodel which allows to define arbitrarily complex mappings between the abstract and concrete syntax of visual DSMLs. As a result, powerful visual abstractions can be introduced to the graphical representation, i.e. a single graphical element may represent (abbreviate) a complex fragment of the underlying abstract syntax.

Obviously, such complex mapping models between the abstract syntax model and the diagram model introduces synchronization problems between the two models. Complex changes in the concrete syntax model (caused by editing operations) need to be immediately reflected in the underlying abstract syntax, and changes in the abstract syntax (e.g. caused by running background model transformations) would have a non-trivial effect on the concrete syntax model. For this purpose, mapping models will be processed with event-driven incremental transformations (Chapter 5), which continuously run in the background to immediately react to complex, non-atomic changes in (any of) these models in an incremental way. These reactions can be designed by relying on the high-level model transformation language of Viatra2 (Section 4.1 and Section 5.3).

Unsurprisingly, the handling of arbitrarily complex mappings between the abstract and concrete syntax have some architectural impact on the underlying DSM framework as well. Therefore, we demonstrate the practical feasibility of the approach using the ViatraDSM framework (Chapter 3).
6.1.2 Contributions

The main contributions of this chapter are, therefore, the following: (1) we propose a mapping model which allows to define arbitrarily complex mappings between the abstract syntax and the diagram model; (2) we demonstrate how live transformations can support to maintain the coherence of these models, (3) we provide (an overview of the) tool support. These concepts will be demonstrated by developing a simple still representative domain-specific modeling language.

Taking a traceability viewpoint, our proposal combines explicit traceability (based on modeling links, see Section 5.1.4) with implicit traceability (when traceability is provided implicitly by a live model transformation transformation running in the background as a daemon). In fact, it is a design decision (depending on the application domain and traceability requirements) how to balance between the two approaches. In this work, we use generic, declarative mapping models for explicit traceability (which can be reused for capturing traceability between other modeling languages). Then one-to-one mappings between source and target elements are handled by generic model transformation rules. However, for more complex (arbitrary m-to-n) synchronizations, we use designated (domain-specific) transformation rules.

6.1.3 Structure

The rest of this chapter is structured as follows. Section 6.2 summarizes main concepts of developing domain-specific modeling languages, and using their corresponding editors. To better motivate our work, we give an overview of the state-of-the-art Eclipse Graphical Modeling Framework (Section 6.2.3), and identify its architectural problems and limitations. Section 6.4 presents our solution for the generic synchronization of the abstract and concrete syntax of DSMLs using mapping models and live transformations. Section 6.6 provides a brief overview on implementation details. Related work is assessed in Section 6.7, and finally, Section 6.8 highlights additional applications of our mapping model and synchronization techniques.

6.2 Challenges of model synchronization in graphical editors

In this section, we use the Eclipse Graphical Modeling Framework as a modern, state-of-the-art domain-specific language engineering environment as the problem context. It is important to note that the ideas and issues explained are not specific to GMF, in fact, they represent a generalization of experience gathered in designing and implementing custom domain-specific languages with various technologies.

6.2.1 Design of Domain-specific modeling languages

In domain-specific visual language design, the three most important design aspects are the following:

- Abstract syntax specification, which is typically carried out using metamodeling. The basic notions of the language (model elements) and their relations (associations) are defined in a mathematically precise way, with structural constraints (e.g. to express containment relations, or type correctness for associations), multiplicities and implicit relationships (such as inheritance, refinement).

- Concrete syntax specification targets the actual visual appearance of the language, assigning a visual symbol to those language elements which are to be represented on diagrams.
• **Language constraints** are frequently also needed, to express correctness criteria that are cannot be specified using metamodeling (e.g. attribute value validity intervals, or complex structural well-formedness rules that involve multiple model element configurations).

Early domain-specific modeling tools such as MetaCase’s MetaEdit+ [Met] derive the structure of the graphical representation from the abstract syntax, as notation definitions are assigned directly for each abstract syntax model element. This is suitable for simple languages with a few element types, however, in today’s practical applications, language metamodels are becoming increasingly large and complex. As a result of the mapping approach, the complexity is propagated into visual diagrams.

Tackling this visual complexity is a major challenge in designing domain-specific modeling languages on the right level of abstraction [SFFG10], which simultaneously provides (i) intuitive graphical syntax without unnecessary details, and (ii) an abstract syntax close to the concepts of the domain.

### 6.2.2 Architectural overview of DSMs

A straightforward strategy to balance abstraction with expressive power is to separate abstract and concrete syntax representations. Essentially, this approach treats the visual notation as a separate language with its own element types, attributes and relations, on an additional modeling layer. For two dimensional graph-like languages (as most visual languages are conceptualized), this visualisation grammar is derived from a core diagram metamodel, which contains attributed nodes and edges. By refining these concepts to specific model elements, the structure of the concrete syntax may be elaborated; visual appearance is specified by traditional design tools as previously.

![Conceptual overview of domain-specific editors](image)

Figure 6.1: Conceptual overview of domain-specific editors

In graphical DSMs, the model is typically manipulated by the designer using a graphical editor over the concrete syntax. Changes initiated in the concrete syntax are immediately propagated to the abstract syntax model. However, in many DSMs, there are certain model elements, which are not visible to the user in the concrete syntax, thus they need to be manipulated directly in the abstract syntax (e.g. by using a Property sheet). Moreover, modern DSM may offer multiple visualizations (diagrams, textual notation, hierarchical overviews) of the same abstract syntax model. In this case, a change in one concrete syntax triggers a change in the abstract syntax, which needs to be reflected
6.2. CHALLENGES OF MODEL SYNCHRONIZATION IN GRAPHICAL EDITORS

instantaneously in the other visualizations. Finally, many complex model manipulations (for model analysis or model transformations) are carried out directly on the abstract syntax, and the result of their execution needs to be reflected in all concrete syntax views preferably immediately. All these scenarios highlight that bidirectional synchronization of various models of the abstract and concrete syntax is a major challenge for DSMs.

For this purpose, a modeling environment typically offers a hybrid view of the model space. Since the user is working with two separate notations of the same model, synchronization has to be done on-the-fly. As abstract and concrete syntax models are stored in separate modeling layers, the solution is a model-to-model synchronizer which maintains both representations and maps changes symmetrically.

6.2.3 The GMF approach and its limitations

As a state-of-the-art environment, the Eclipse Graphical Modeling Framework follows this design pattern.

At run-time, GMF maintains two distinct model instances: the abstract syntax models conforming to the ECore metamodel, and a Notation model conforming to a built-in Notation metamodel. Notation models correspond to diagrams and contain only visualisation-specific information. The user is interacting with a parameterized, but still generic Notation model editor, where these parameters contain information on how diagrams can be manipulated and mapped to domain models.

GMF automatically performs the mapping as the user is editing the model, according to a built-in semantics. Model synchronization is implemented by using a simple traceability mechanism, where each Notation model element references a corresponding abstract syntax counterpart. This trace model is contained in the Notation model resource, and implements a simple one-to-one mapping (with the exception of labels, which may reference several attributes through special format strings). By this approach, GMF is able to partially separate abstract and concrete syntax representations.

Problems with the GMF approach Unfortunately, in case of advanced applications, severe problems arise due to architecture-level design decisions and limitations of GMF.

1. The GMF semantics is restricted to one-to-one mappings between model elements of the abstract syntax and their graphical representation. In case of complex modeling languages (like AUTOSAR), abstraction capabilities of the concrete syntax would be advantageous e.g. to allow a graphical notation to abbreviate more than a single element in the underlying concrete syntax.

2. It is impossible in GMF to fully separate the abstract syntax and the concrete syntax of a language. In fact, GMF imposes some (hidden) meta-constraints on the abstract syntax such as the existence of a Diagram notion in the metamodel, or the connection classes providing navigation from both directions. Additionally, as the abstract syntax model is not independent from GMF’s visualisation, significant development efforts are required to tailor existing Eclipse Modeling Framework (EMF [?]) models to be GMF-compliant.

Overcoming these problems frequently requires significant programming effort, which is specific to the modeling language itself. For instance, in the case of the industry-standard AUTOSAR metamodel, developing a GMF based editor requires the creation of a new GMF-compliant metamodel (for the sublanguage which is planned to be visualized), with expensive ad hoc synchronizers in-between (as illustrated in Figure 6.2).
6.3 Modeling the Abstract and Concrete Syntax

6.3.1 Novel architecture for synchronizing abstract and concrete syntax

In this work, we propose a novel solution to completely separate the abstract syntax and concrete syntax of a graphical modeling language with arbitrary mapping between them using advanced trace-ability models and live model transformations. Our approach will be demonstrated using the ViatraDSM framework (Chapter 3) for developing domain-specific modeling languages using a novel underlying architecture (depicted in Figure 6.3). The essence of the solution can be summarized as follows.

- A general mapping model is used to connect elements of abstract and concrete syntax, which significantly extends the capabilities of GMF mappings.
- Metamodel-tagging is used for the abstract syntax (conceptual metamodel) of the modeling language, which eliminates the need for introducing a separate conceptual sub-language (as frequently necessitated by GMF).
6.4. GENERIC ABSTRACT-CONCRETE SYNCHRONIZATION WITH MAPPING MODELS AND LIVE TRANSFORMATIONS

- A general model transformation language is used to specify more complicated mappings between the abstract and concrete syntax on a high-level of abstraction (namely, as a model transformation solution instead of pure Java code).

- Live model transformations are used as an execution mechanism driven by changes in the underlying models to achieve high performance even for large models.

- A mapping library is provided as a guideline to accelerate the implementation of abstract-concrete syntax mappings.

6.3.2 A synchronization problem between abstract and concrete syntax

Example 18 (Abstract-concrete syntax synchronization in the Petri net DSML) As a demonstrating example, we will construct an advanced domain-specific modeling environment for the Petri net language (using the architectural considerations discussed in Section 6.2.2 and Section 6.3.1). The editor provides a graphical concrete syntax representation for Petri net graphs, with support creating Place and Transition nodes and In/OutArc edges (ensuring syntactic correctness while editing, i.e. an OutArc can only start at a Place). Graphical attributes (such as token count for Places and arc weight for Arcs) can be edited through a standard property editor. As a framework for the example, the ViatraDSM domain-specific environment (Chapter 3) based on the ViTRA2 model transformation system will be used.

Our example editor implements the abstraction shown in Figure 6.4. It involves hiding token instances and displaying the number of associated Tokens as a numeric label inside the Place instance, whereby a derived model property (the number of token instances) is mapped to a diagram element property (tokenCount).

![Figure 6.4: Model layers in the Petri net editor](image)

6.4 Generic abstract-concrete synchronization with mapping models and live transformations

In this section, we combine our event-driven transformation approach (Chapter 5) with the trace metamodels (Section 3.4.2.2) to provide a generic, metamodel-driven transformation approach for the on-the-fly synchronization and tracing of abstract and concrete syntax representations of a graphical domain-specific language. By our approach, the two modeling layers can be fully separated, making arbitrary visualisation abstractions possible.
First, we describe the core cases of simple \textit{one-to-one} model synchronizations (Section 6.4.1). These \textit{synchronization primitives} build on reference mapping models to define graph transformation rules for handling model creation, deletion and attribute updates.

Based on these primitives, we propose a \textit{Mapping Library} (Section 6.4.2), which uses the metamodels introduced in Section 3.4.2.2. Our generic approach is applicable to any domain, as it follows correspondence pairs encoded in a \textit{mapping metamodel}. This library replicates GMF’s mapping functionality for ViatraDSM editors, but also provides a flexible starting point for further mapping customization in order to allow language engineers to go beyond GMF’s mapping capabilities.

In Section 6.5, we demonstrate that by combining the basic techniques from Section 6.4.1 with the generic approach in Section 6.4.2, a language engineer may create custom mappings easily. As a proof-of-concept demonstrating example, we rely on models introduced for the abstraction mapping outlined in Example 14.

### 6.4.1 Trace models in event-driven transformations

In a synchronization scenario, a source and a target model (or an entire source graph and target graph) are present, which have to be kept synchronised at all times. Note that usually this is not merely a batch model transformation from a source model to new target models, both models evolve concurrently. Every modification on the source model has to be followed on-the-fly with the relevant modification on the target model. Furthermore, the consistency of the models has to be maintained, so every change in the target model which is relevant to the source model, has to be handled as well. Usually these two models are fully separated from each other. This means that a model element in the source model may not have a direct link (reference) to its corresponding model element in the target model. Moreover, multiple elements from the source model can be related to a single target element, hence it is necessary to use trace models which connect source and target models.

In the following general example, we outline the core cases of the \textit{source} $\rightarrow$ \textit{target} synchronization scenario. In Figure 6.5a, a consistent state of the model space is shown, where a \textit{SourceElement} instance is connected to a \textit{TargetElement} instance by a trace model instance of type \textit{ReferenceElement}. This configuration expresses the correspondence relationship between the source and target model elements.

In the following scenario, we define a live transformation fragment which detects that a new source model element has been created and creates the corresponding target model element (Figure 6.5b). The \textit{Neg} area marks a \textit{negative application}.
condition over the reference and target model elements. Therefore, a rise trigger will fire when a new source model element which does not yet have a corresponding pair in the target model, has been created. The action sequence of the graph trigger will then proceed to create exactly those elements which are included in the negative application condition (as indicated in Figure 6.5b by the New keyword).

Detecting element deletion in the source models As long as the model space is kept consistent, every SourceElement has a TargetElement pair, and an appropriate ReferenceElement with its relations. As a consequence, deletions in the source model hierarchy can be handled by a graph transformation trigger shown in Fig. 6.5c. After deleting a SourceElement, a ReferenceElement remains, without the R-S relation.

We again use a negative application condition (marked Neg) in a rise trigger to detect a new occurrence of a such a ReferenceElement–TargetElement stub (Pre indicates that both the ReferenceElement and TargetElement instances are included in the precondition). In the action sequence, the graph transformation rule will proceed to delete the ReferenceElement and the TargetElement instances.

Detecting attribute updates in the source models While attribute value changes can be detected using techniques described in Section 5.3.2.3, detecting a change with respect to the last synchronized value involves storing values in the trace models. As shown in Figure 6.5d, we define an attribute check condition on the value equality of attributes stored in the source and trace models. In this way, the pattern matcher will detect when an attribute update has occurred. Note that the action sequence is omitted from Figure 6.5d since it may be domain-specific (e.g. the trigger may fire an attribute update in the target models).

6.4.2 Generic abstract–concrete syntax mapping

By combining the basic techniques described in Section 6.4.1 with trace metamodelling and modeling as shown in Section 3.4.2.2, we show a generic approach to abstract-concrete synchronization. Conceptually, this approach establishes a Mapping Library to provide a GMF-like one-to-one mapping facility. Note that due to space constraints, we focus on illustrating the core ideas; the complete implementation is available as part of the standard Viatra2 software distribution.

A metamodel-driven generic transformation takes a specification metamodel as an input to determine rules that describe how the transformation should be performed. In the trace metamodel (Figure 3.22), corresponding source and target model element types are connected with MappingElement types to indicate, for instance, that any given Place instance should be mapped to a PlaceFigure instance and vice-versa. Note that the core mapping metamodel (Figure 3.21 and 3.16) allows assigning multiple abstract syntax elements to a concrete syntax element: for instance, (i) multiple Nodes and Edges may be assigned to a MappingElement, and (ii) multiple MappingElements may be assigned to a DiagramElement) supporting a many-to-many mapping semantics. In this example, we demonstrate a more simple, one-to-one correspondence which is analogous to GMF’s capabilities.

The graph transformation triggers below are presented in a compacted notation. In the figures, abstract syntax model elements appear on the left (with the AS_ prefix for pattern variables), while concrete syntax elements appear on the right (CS_).
6.4.2.1 Capturing types in graph patterns

Figure 6.6 shows the generic `existsInMetaModel` subpattern which demonstrates how graph triggers can be defined to match any domain. This subpattern matches domain metamodel elements (subtypes of the core `NodeFigure`, `NodeMapping` and `Node` elements; subtyping is denoted shortly by square brackets) and provides pattern variables (CS_TYPE, TR_TYPE, AS_TYPE in Figure 6.6) which pass type information regarding contextual information captured in the mapping metamodel. For instance, these pattern variables may take model references to `PlaceFigure`, `PlaceMapping`, and `Place` as values.

```plaintext
pattern existsInMetaModel(AS_TYPE, CS_TYPE, TR_TYPE) =
{
  // refinement of core domain metamodel
  supertypeOf(MetaNode, AS_TYPE);
  // refinement of core mapping metamodel
  supertypeOf(MetaNodeMappingElement, TR_TYPE);
  // connecting relationships
  relation(_, TR_TYPE, AS_TYPE);
  relation(_, TR_TYPE, CS_TYPE);
}
```

Figure 6.6: The `existsInMetaModel` generic graph pattern

6.4.2.2 Detecting creation in the concrete syntax

In Figure 6.7a, the `linkNodeFigure` trigger is presented. This trigger creates domain-specific `Nodes` for every `NodeFigure` which is created by the user during model editing (the direct type of `Nodes` and `NodeFigures` is passed as pattern variables from Figure 6.6). Since the concrete syntax metamodel allows for creating concrete syntax nodes in two contexts (as top nodes placed directly on the diagram and as sub nodes of a container node), the live transformation sequence has two modes of operation. Correspondingly, the precondition pattern of the `linkNodeFigure` trigger (Figure A.1) is an OR pattern, which defines a logical disjunction for each of the cases.

Both subpatterns share the same structure; a negative application condition (marked with dark grey) ensures to match against concrete syntax model elements, which do not yet have corresponding mapping and abstract syntax elements. Note that `NodeFigure`, `Diagram`, `Hierarchy`, `TopMapping`, `NodeMapping` are indirect, generic types in this case, the direct domain-specific type is only relevant for the concrete, mapping and abstract syntax nodes (CS_NODE, TR_NODE_MAP, AS_NODE are tagged with type values CS_TYPE, TR_TYPE, AS_TYPE respectively).

The trigger creates these missing elements, both the abstract syntax node and the mapping node with connecting relationships, similarly to the creation synchronization primitive in Section 6.4.1.

The pattern contains generics to express a general type-instance relationship between concrete syntax nodes (CS_NODE) and their types in the domain-specific concrete syntax metamodel (CS_TYPE), with similar constraints for abstract syntax (AS_NODE and AS_TYPE) and trace models (TR_NODE_MAP and TR_TYPE). Hence, this transformation can adapt to any domain, since the type information in the pattern will be used to create type-correct model instances in the action sequence (AS_TYPE, REL_TYPE, TR_TYPE).
6.4. GENERIC ABSTRACT-CONCRETE SYNCHRONIZATION WITH MAPPING MODELS AND LIVE TRANSFORMATIONS

6.4.2.3 Detecting deletions

Figure 6.7b shows the deleteHandling trigger, which demonstrates how to detect deletion in both abstract and concrete syntax models.

This rise trigger also references the generic subpattern in Figure 6.6 for type information. We use a disjunctive OR-pattern to handle the following cases:

- The first OR-subpattern corresponds to the case where a concrete syntax node has been deleted. This is signaled by the appearance an abstract syntax node with its related mapping element without a related concrete syntax node. As a reaction, the mapping model element has to be deleted along with the related abstract syntax element, in parallel with the GMF mapping semantics (“delete from model” operation).

- In the second case, we define an OR-subpattern which corresponds to the case when an abstract syntax element is deleted. This event is signaled by the appearance of a concrete syntax node – mapping model node pair without a connected abstract syntax node. As a reaction, the concrete syntax element is be deleted from the graphical representation along with the
mapping element. Note that extending the basic techniques to bidirectional synchronization is straightforward, since symmetries can be easily exploited in pattern definitions.

6.4.3 Summary

In Section 6.4.2, we have highlighted the foundations of a generic Mapping Library, which leverages our mapping metamodel and live transformation technology to provide a general solution for the incremental synchronization of abstract and concrete syntax models. By combining simple techniques, our approach provides a GMF-like one-to-one mapping semantics, which works for arbitrary domains and is specified using a high abstraction level transformation language.

6.5 Arbitrary abstract–concrete syntax mapping

In practical applications, the need for a custom mapping frequently arises, where a mapping rule framework is needed for a special abstraction, e.g. to simplify the visualisation of a complex modelling language. In state-of-the-art frameworks such as GMF, the language engineer is stuck with the default options provided, and customization beyond those requires extensive programming, which is only possible when the application programming interface allows for a straightforward programmatic hook at the right places.

In contrast, our transformation-driven approach provides extensibility and customization at a significantly higher level of abstraction. By simply defining custom graph triggers or overriding the ones provided by the Mapping Library, the language engineer may customize abstract-concrete syntax mappings using the techniques shown previously.

Example 19 (Petri net token abstraction) In the following example, we use the context of the Petri net case study (Section 6.3) to demonstrate a common mapping abstraction, where the visualisation layer presents the number of model elements of a certain type (tokens) as a simple numeric attribute, instead of assigning a graphical diagram element to each (Figure 6.4).

This custom synchronization transformation has to perform two tasks:

- **concrete → abstract syntax synchronization**: whenever the attribute value in the concrete syntax model changes (e.g. the user changes its value through the GUI), the appropriate number of Token instances should be assigned to the Place instance the owner of the changed attribute was mapped to (by creating new tokens or deleting existing ones).

- **abstract → concrete syntax synchronization**: symmetrically, when a new Token instance is assigned to the Place, or a previously existing one is deleted, the attribute value of the PlaceFigure must be updated accordingly.

6.5.1 Precondition patterns for detecting changes

**Creation**  In practical applications, a chain of triggers may be used to execute multiple incremental updates. For instance, after a Token instance has been added by the user, the system may execute a trigger which automatically connects it to a Place (Figure 6.8, tokenAdded). After tokenAdded has fired, another trigger similar to Figure 5.9 (tokenConnected) updates the numberOfTokens array stored in the execution context.
6.5. ARBITRARY ABSTRACT–CONCRETE SYNTAX MAPPING

@Trigger (sensitivity = rise)
gtrule tokenAdded () =
{
    precondition pattern token (T) = {
        Token (T);
    }
    action {
        println("A token was added: " + name (T));
        // action: find a place and connect the
        // unconnected token to it
        choose P with find place (P) do
            new Place . tokens (_, P, T);
    }
}
asmfunction numberOfTokens / 1;

@Trigger (sensitivity = rise, priority = 1)
gtrule tokenConnected () =
{
    precondition pattern connectedToken (P, T) = {
        Place (P);
        Token (T);
        Place . tokens (_, P, T);
    }
    action {
        update numberOfTokens (P) = numberOfTokens (P) + 1;
    }
}

Figure 6.8: Trigger to handle the addition of Tokens

Deletions To detect deletions, a trigger for the same precondition pattern as used in Figure 6.8 can be used in fall mode. In this case, a when-clause is used to filter the case when the match set loss occurred because of the deletion of a model element referenced by the T variable (Figure 6.9).

@Trigger (sensitivity = fall, priority = 1)
gtrule tokenRemoved () =
{
    precondition find tokenAdded . connectedToken (P, T)
    action {
        // only act if token T has been lost (deleted)
        when (delete (T)) seq {
            update numberOfTokens (P) = numberOfTokens (P) - 1;
        }
    }
}

Figure 6.9: Handling token deletion

Detecting changes in the concrete syntax model We use the technique shown in Section 6.4.1 to define a precondition pattern for detecting changes in attribute values in the concrete syntax model (Figure 6.10).

Detecting changes in the abstract syntax model In order to trace the amount of tokens assigned to a given Place instance, we may use trigger precondition described in Figure 6.11 in triggers similar to Figure 6.8 and 6.9. They update the global numberOfTokens array whenever Token instances are created and deleted; by combining that technique with the one we used in Figure 6.10 we define the precondition pattern in Figure 6.11 which includes a check condition on the equality of the value
Chapter 6. Synchronization Between Abstract and Concrete Syntax

```plaintext
pattern attributeTrace(CS_NODE, CS_ATTR, TR_MAP_NODE) =
{
  // concrete syntax
  PlaceFigure(CS_NODE);
  TokenCount(CS_ATTR);
  PlaceFigure.tokenCount(_, CS_NODE, CS_ATTR);
  // trace model
  PlaceMapping(TR_MAP_NODE);
  // check condition
  check (value(TR_MAP_NODE) == value(CS_ATTR));
}
```

Figure 6.10: Trace pattern for an attribute value change

stored in the trace model and the one cached in the global array. By defining a fall trigger with this pattern, the system may detect when the user has changed the number of Token instances assigned to a Place instance whose concrete syntax mapping is maintained.

```plaintext
pattern tokenCountTrace(AS_NODE, TR_MAP_NODE) =
{
  // abstract syntax model
  Place(AS_NODE);
  // trace model
  PlaceMapping(TR_MAP_NODE);
  nodeMapping(_, TR_MAP_NODE, AS_NODE);
  mappings(_, AS_NODE, TR_MAP_NODE);
  // check condition
  check (value(TR_MAP_NODE) == numberOfTokens(AS_NODE));
}
```

Figure 6.11: Pattern for detecting attribute value changes in the abstract syntax

As an alternative, one may use the new pattern language feature of the ViATRA2 system which supports assigning a cardinality constraint to a subpattern. Such a constraint defines the exact number of matches a pattern might have, so using a global array variable is no longer necessary to detect changes in the occurrence count of a model element type (Figure 6.12).
6.5. ARBITRARY ABSTRACT–CONCRETE SYNTAX MAPPING

pattern placeToken(P,T) =
{
  Place(P);
  Token(T);
  Place.tokens(_,P,T);
}

pattern tokenCountTrace_advanced(AS_NODE, TR_MAP_NODE) =
{
  // abstract syntax model
  Place(AS_NODE);
  find placeToken(AS_NODE,_,T)# NumTokens;
  // trace model
  PlaceMapping(TR_MAP_NODE);
  nodeMapping(_,TR_MAP_NODE,AS_NODE);
  // check condition
  check (value(TR_MAP_NODE) == NumTokens);
}

Figure 6.12: Trace patterns for cardinality

6.5.2 Solution by graph triggers

Finally, in Figure A.2, we combine the two precondition patterns into a disjunction to provide a complex precondition pattern for the synchTokens trigger. The pattern call to mappedPlaceFigure is only used to ensure that the entire precondition pattern configuration corresponds to exactly one abstract-concrete-trace tuple. In the action sequence, we use when-clauses to distinguish between the two operation modes. When synchronizing the attribute value change, the transformation computes the difference \( \text{Diff} \) in the number of tokens to the previously known value stored in the trace model \( \text{value(TR_MAP_NODE)} \) and proceeds to call a sub-routine which creates or deletes the necessary amount of tokens \( \text{addOrRemoveTokens} \). After that, the trace model is updated.

In the other case, when the abstract syntax model was changed, the value of the \( \text{numeberOfTokens} \) array is simply copied into the concrete syntax attribute \( \text{setValue(TR_MAP_NODE, numberOfTokens(AS_NODE))} \). Finally, the trace model is updated with the new information.

A sample execution sequence In Figure 6.13, a sample execution sequence of the synchTokens trigger is shown (note that edge types have been omitted for the sake of simplicity). In Phase 1, the model is in a consistent state, where place \( P_0 \) contains a token \( T_0 \), and this fact is reflected in the diagram model as a token count attribute value of 1, stored in \( TC \).

Next, we follow the scenario where the user adds a token to \( P_0 \) (Phase 2). As a reaction, the pattern matching RETE network assigned to the connectedToken graph pattern signals a new match, and the tokenConnected trigger is fired (Figure 6.8). As the slot assigned to \( P_0 \) in the numberOfTokens global array is updated by trigger, the RETE network again signals a match set loss in the tokenCountTrace graph pattern (Fig. 6.11), which in turn fires the synchTokens trigger (Figure A.2). As a result, first, the attribute value in the concrete syntax is updated (Phase 3), and finally the reference value in the trace model is modified (Phase 4).

6.5.3 Summary

In the current section, we have demonstrated the flexible extensibility of our core approach. By combining basic mapping techniques from the Mapping Library, the language engineer is able to specify
CHAPTER 6. SYNCHRONIZATION BETWEEN ABSTRACT AND CONCRETE SYNTAX

(a) Phase 1
(b) Phase 2
(c) Phase 3
(d) Phase 4

Figure 6.13: Petri net synchronization execution phases

an arbitrary mapping between abstract and concrete syntax models. This way, the domain-specific visualization and editing framework can be directly adapted to any abstract syntax metamodel, without the need of constructing an intermediate language (Figure 6.3).

Our approach can be extended to scenarios where the same abstract syntax model is mapped to multiple diagrams, which is a frequent requirement in advanced modeling environments. In that case, for $n$ diagram types, $n$ (bi-directional) mapping transformations have to be developed; whenever a change is made in one of the diagrams, the change is propagated to the abstract syntax and then automatically propagated further to those diagrams where the affected abstract syntax elements are displayed. This way, the consistency across multiple diagrams can be automatically preserved.

6.6 Implementation details

The authors have developed a complete implementation of the approach described in Section 6.4, which is adapted to the ViatraDSM domain-specific language engineering framework. The Mapping Library (Figure 6.3) consists of generic live transformation programs written in the VIATRA2 Textual Command Language, and are capable of facilitating a two-way, one-to-one correspondence mapping based on the trace metamodels described in Section 3.4.2.2.

Along with this implementation, we have also provided a prototype VIATRA2 import facility, which is able to process GMF specification models (.ecore, .gmfgraph, .gmfmap). The importer generates a domain-specific graphical editor for the ViatraDSM framework, which is functionally equivalent to the GMF editor.

- first, the EMF .ecore metamodel is converted into a ViatraDSM abstract syntax metamodel conforming to the Core metamodel.
- next, the GMF .gmfgraph metamodel is converted into a ViatraDSM concrete syntax metamodel conforming to the Core Diagram metamodel.
- finally, the GMF .gmfmap metamodel is processed to create a ViatraDSM mapping metamodel conforming to the core mapping metamodel.
6.6. IMPLEMENTATION DETAILS

(a) Petrinet domain in EMF

(b) Petrinet domain in ViatraDSM

Figure 6.14: Importing the Petri net domain metamodel to Viatra2

Figure 6.15: The Petri net editor in ViatraDSM
The converted editor (the working example Petri net editor is shown in Figure 6.15) works similarly to the original GMF editor; the user can place the same elements on the diagrams and edit the same attributes. However, ViatraDSM allows direct access to the full abstract syntax model (as shown in the Outline view on the right in Figure 6.15), so it can be manipulated independently of the concrete syntax. The Mapping Library provides bi-directional synchronization between the two representations; it can be easily extended at run-time so that a custom mapping can be developed rapidly.

![Figure 6.16: Processing the graphical definition](image)

### 6.7 Related work

In this related work section, we provide a brief evaluation of leading commercial and academic initiatives in the field of domain-specific modeling frameworks, with a special focus on support for abstract–concrete syntax synchronization and model transformation support.

The limitations on the application of GMF to practical DSMLs has been summarized recently in [KRPP09, KRA+10], describing the EuGENia tool. This approach is also focused on the application of model transformation technology to abstract GMF’s complex internal details, but uses transformations at design time rather than at runtime (as in our case).

Our application of traceability is different from the traditional traceability applications to requirements management and tracking in model driven scenarios, described in many papers (e.g. [CR03, WJSA06, DKPF09, PDK+10, DMKPF10]). In the current paper, traceability models are used to link two representations of the same modeling language (namely, abstract and concrete syntax) together, to drive a bi-directional synchronization transformation (conceptually similar to e.g. Fondement’s work [MFF+06]). As emphasised, our contribution is two-fold: (i) we use event-driven live transformations to facilitate the automatic generation of trace models and the execution of the mapping, (ii) we use a generic approach to trace metamodels which enables the designer to choose how much information is contained in the trace models and how much mapping logic is (implicitly) implemented in the transformations themselves.

Recently, [Mar08] emphasized the use of weaving models as a special kind of correspondence models to semi-automatically derive model transformation rules for model synchronization. The authors present a metamodel-based method that exploits metamodel data to automatically produce weaving models in the AMW System. The weaving models are then derived into model integration transformations.

Fondement’s work shares a lot of concepts presented in our paper: a complete mapping metamodel with semantics is provided in [MFF+06] to support arbitrary mappings between abstract and concrete syntaxes of textual DSLs (transformation execution is carried out with the Kermeta tool).
6.8. SUMMARY

(a) gmfmap

(b) ViatraDSM mapping

By his approach, mapping semantics is precisely defined for the model elements (e.g., sequence, alternation, iteration, template substitution rules); in our approach, the designer is free to choose how much semantic information is included in the mapping model and how much is implicitly defined in the transformation rules. Also, it is important to note that while the mapping transformations are incremental, they are not live, but executed in a recursive descent-type batch execution scheme.

6.8 Summary

As the core contribution of this chapter, we presented a new approach for constructing syntax-driven domain-specific graphical editors. Building on this infrastructure, we provide high level support for the specification and efficient execution of live transformations which seamlessly maintain correspondence between completely separated abstract and concrete syntax representations. Our approach provides a scalable solution in terms of complexity, since language engineers can build on a generic Mapping Library to create custom mapping rules which focus strictly on those cases where customization is really necessary.

In this work, we focus on solving model synchronization problems in DSM environments. It is important to note that our approach has been implemented and tested in more complex case
studies than the Petri net example of the thesis. We successfully applied the proposed mapping models in other application scenarios such as incremental well-formedness constraint evaluation where mapping models are used to indicate model contexts where a particular constraint is violated [20]. The approach has also been applied to interactive model execution and design-time discrete simulation of DSMLs, elaborated as case studies in Section 7.

The results of this chapter are formulated as thesis contributions as follows:

2.2 I elaborated a technique for the complete separation of abstract and concrete syntax of domain-specific modeling languages (Challenge 2) based on generic event-driven model transformations [7,23,24] (Section 6.4 and 6.5).

This approach overcomes a significant limitation of state-of-the-art language engineering frameworks, where the concrete syntax representation is tied to the structure of the abstract syntax model, and does not allow strong abstractions that may be needed for simplified visualisation.

The technique is based on event-driven model transformations of Chapter 5. With the aid of generic specification mapping models conforming to a mapping metamodel, this technique can be used to model correspondence relationships between metamodels of abstract and concrete syntax, and automatically perform the mapping on instance models by a library of customizable graph triggers.
2.4 I proposed efficient implementation techniques for the approach. The prototype implementation is available in the ViatraDSM tool [26] (Chapter 3).
Chapter 7

Model simulation in domain-specific modeling languages

Contributions of the current chapter The current chapter extends the event-driven execution semantics for model transformations to support user-driven, interactive model execution (Challenge 3) at design time, within the modeling environment itself (Progress map 4). By this approach, the discrete execution rules (transitions between implicit states) are specified as event-driven transformations, and their execution may be guided directly by the user (by selecting from the enabled rule-context pairs), or by an automated (stochastic) scheduler (Contribution 5). The approach is implemented in the ViatraDSM environment, to support the design-time execution, debugging and rapid prototyping of dynamic DSMLs.
7.1 Overview

Domain-specific modeling languages are frequently used to model systems that interact with their environment in various ways: interaction may occur between software components, (multiple) user(s), or the physical surroundings in e.g. embedded systems. Such requirements manifest at design time in the usage of dynamic modeling languages that allow high level specifications of such interactions. Moreover, dynamic semantics are not only specified, but – if possible – also executed in development tools, in order to allow debugging and experimentation with the models themselves (instead of e.g. generating code and testing the software directly in the target environment).

Model simulation is typically carried out using dedicated simulators (such as MATLAB). These tools support high-performance analysis that allows the analysis of critical system properties such as reliability and performance, prior to actual implementation and deployment. Unfortunately, these powerful features may come at a high implementation cost, as these specialised tools rely on complex specification languages, may need expensive hardware/software environments and, most importantly, as the analysis is performed in their special, mathematical domain, the results may be hard to translate back to the original modeling language. For this back annotation of e.g. simulation runs (dynamic traces) and analysis results, model transformations are frequently used.

In many practical applications, the dynamic semantics of DSMLs do not require elaborate, high-precision mathematical computations that tools like MATLAB offer. Instead, they can be mapped to simple dynamic languages such as state machines, Petri nets or data flow networks (as emphasized in e.g. [KRDM+10]). These support discrete simulation scenarios where (i) the dynamic state of the system can be expressed in terms of discrete (model-based) configurations, and (ii) state transitions occur at discrete time intervals. This category of systems belong to Discrete Event System Specification (DEVS) [Zei84] that have been used in the past decades in a wide spectrum of practical model-based simulation applications.

Within this framework, many model and graph transformation tools [dLV02, LLMC04, EEHT05, GBG+06, Ren04a, KW07, KPP08] may be used as alternatives or complementary technologies to dedicated model simulation tools [SV08]. As a common approach, these tools allow to use the underlying transformation engine directly to 'animate' domain-specific models at editing time, which makes the rapid prototyping of dynamic languages feasible. However, such tools are not intrinsically targeted at language engineering, so they are also typically used as complementaries to DSM frameworks, which, in turn, leads to the same back-annotation problem encountered with dedicated tools.

Contributions In this chapter, we present a novel domain-specific model simulation framework for discrete domains, that combines event-driven transformations to support both automatic and interactive dynamic semantics execution, within a fully featured language engineering environment. As a core contribution, our tool – based on incremental pattern matching – provides high performance simulation that scales to large model sizes, which allows its direct application in exploratory system simulation (instead of dedicated simulators). The simulator has also been adapted to support stochastic execution, where probability distribution functions can be assigned to rule-match pairs that govern their scheduling. This work is discussed in detail in Appendix B.

7.2 Model simulation by event-driven transformations

Discrete simulation processes are useful to cover a wide range of dynamic DSMLs, such as token games, (discrete) data flow networks, state automata with arbitrary memory extensions etc. As model
7.2. MODEL SIMULATION BY EVENT-DRIVEN TRANSFORMATIONS

Elements can be freely created and deleted (birth-and-death processes), this approach supports a flexible modeling framework for dynamic systems where not only a designated subset of the model (such as token distributions in Petri nets or active states in state machines), but the entire structure can evolve as time progresses (dynamic reconfiguration).

Definition 32 (Discrete simulation process) A discrete model simulation process over a model space (Def. 12) is a trajectory of model changes, equivalent to a sequence of transactions (Def. 29): \( \text{MSP} = \overrightarrow{T} \). In simple terms, such a simulation process can consist of the following model changes in-between state transitions: (i) creation/deletion of model elements, (ii) attribute value changes (changes in the value VPM function), (iii) structural changes (changes in src, trg, supertypeOf, instanceOf VPM predicate values).

To support this flexible specification approach, dynamic semantics of domain-specific modeling languages are described by simulation rules that are essentially event-driven graph transformation rules (Def. 27). Simulation rules have two states that are evaluated at discrete time intervals by the execution mechanism: they can either be (i) enabled (meaning that they can be executed on the model to transfer the system to the next state), or (ii) disabled (meaning they cannot be executed).

Definition 33 (Model simulation rule) Similarly to event-driven rules, simulation rules are defined by enabledness conditions that are event definitions (Def. 25) and action sequences augmented with conditions (Def. 26). Formally, a model simulation rule \( \text{MSR} \) is defined as \( \text{MSR} = (\text{EDR}) \).

Example 20 (Petri net firing simulation rule) Figure 7.1 shows an example simulation rule for Petri nets. We reuse the code from Figure 5.7 but use the @SimulationRule annotation instead of @Trigger to indicate that this graph transformation rule should be used as a simulation rule. As the precondition, this example calls to the fireable(Tr) graph pattern (Figure 4.2) which (i) defines the execution context of the simulation rule to be individual Transition instances (as Tr is the only parameter of the rule), and (ii) reuses the fireTransition(Tr) ASM rule (Figure 4.13) as the action.

7.2.1 Execution of simulation processes

The execution of a simulation process can be regarded on two abstraction levels: (i) on the model space level, it corresponds to a sequence of model manipulation transactions; (ii) on the simulation level, it corresponds to a sequence of simulation state transitions. The simulation state defines the rule-match pairs that correspond to model contexts where the execution of a simulation rule is enabled: for instance, a particular simulation state in a Petri net \( PN \) may correspond to the case where only transition \( t_1 \) is enabled for firing; another state occurs where both transitions \( t_0 \) and \( t_2 \) are
fireable etc. A *simulation transition* occurs when the execution of a simulation rule for a concrete match is performed. The high-level model for such a simulation process is defined by a *simulation scheme* as follows:

**Definition 34 (Simulation scheme)** A simulation scheme $SIM$ over a model space $MS$ is a structure $SIM = (\{S\}, S_0, \{MSR\}, act)$, where

- $\{S\}$ is a set of states (given by all possible configurations of the underlying model space $MS$),
- $MSR$ is a set of simulation rules,
- $S_0$ is the initial state corresponding to the initial state of the model space $MS$,
- $act : S \rightarrow SMSR \times m$ is a function that gives the set of enabled simulation rule-match pairs enabled in a state.

The execution semantics for simulation schemes is similar to that of event-driven rules (based on queue automata) described in Section 5.2.2.5 (especially Figure 5.6), albeit with a crucial difference: for simulations, *events* are not arbitrary model manipulation transactions, instead the execution follows an implicit state trajectory where each state corresponds to the currently enabled set of rule-match pairs, and a state transition occurs whenever a rule-match pair has been selected (by the user, or an automated algorithm) for execution. In other words, while event-driven transformations are reactive to model changes, simulations are *guided* either interactively or automatically. To define this behavioral difference more precisely in previously established terms: in simulation execution, the *execution mode* (Section 5.2.2.4) can be thought of as a special case of *choose mode* where the choice is either made by the user or an automated algorithm.

**Execution algorithm for simulation** The detailed execution algorithm is illustrated in Figure 7.2. Here, the *WAIT superstate* represents an abstract state where the execution is suspended and is waiting for input on how to proceed. The *POP Queue* (which is a waiting state indicated by grey color) represents the selection of an enabled simulation rule-match pair that can be performed by the user or automatically. When a simulation rule has been selected, it will be executed as with event-driven transformations. In the *EVAL* state, a re-calculation is performed for enabledness conditions, and the model simulation rule (MSR) queue is updated accordingly. *Scheduling rules* here are trivial as no priority etc. needs to be considered, all such logic is implemented in the POP Queue state.

Technically, the system relies on the same supporting infrastructure as event-driven transformations: transaction log, RETE-based incremental pattern matching and delta monitors. The MSR registry is used for managing model simulation rules. It is important to note that model simulation scheme execution may very well interact with the event-driven transformation engine as all manipulation is performed in transactions.

Additionally to user input, the scheduling in the POP Queue state can be determined algorithmically. This can be based on factors such as contents of the model (as in e.g. higher order transformations where the transformation engine can manipulate its own program by making changes to the model space), or probability distribution functions for stochastic simulation purposes. The latter application scenario is described in more detail in Appendix B.

**Example 21 (Petri net simulation)** The most important phases of the simulation process in the Petri net example are illustrated on Figure 7.3.
7.2. MODEL SIMULATION BY EVENT-DRIVEN TRANSFORMATIONS

Figure 7.2: Simulation execution algorithm

Figure 7.3: Petri net simulation phases
1. The simulation begins by the evaluation of all enabledness conditions; for the single simulation rule of the Petri net example, this is done by retrieving the matching set for the isTransition-Fireable pattern (Figure 4.2) from the RETE network. For the sample Petri net in Figure 7.3a, the initial matching set only includes the fork transition. Thus, the initial simulation state has been calculated; since there is only one enabled transition, the user may proceed by firing that transition, or by invoking the automatic execution mode to proceed to the next simulation state.

2. Figure 7.3b shows the sample Petri net after the fork transition has been fired. As the Token model elements have been moved by the simulation rule, the RETE network was automatically updated to indicate that now transitions join and exit are enabled. Since two possible continuation paths exist, this non-determinism may be resolved randomly (in case of automated execution) or by user choice (guided simulation). If the exit transition is chosen, the system will reach a trap state (Figure 7.3c). In a trap state, no enabledness conditions are fulfilled, thus there are no possible continuation paths.

3. As the simulation system is fully integrated into the ViatraDSM domain-specific modeling framework, the user can make various changes as the simulation is being executed. The simulation engine calculates the simulation state incrementally based on the actual model, the user may influence the simulation process by editing the model on-the-fly. In Figure 7.3d, two possible editing actions are shown: the user may re-enable the transition join by either placing a new token into the empty input place (denoted with 1), or delete the input arc (denoted with 2). In both cases, the RETE network is automatically updated following the editing action, moving the simulation system back into a state where join is enabled for firing. This feature is analogous to variable overwriting in program debuggers, however it allows for modifications on the DSM level. The user may also execute model editing which does not influence the simulation state. Fig. 7.3e shows a possible final state for the simulation (when a cycle has been completed) where the user has fired the join transition and removed a token from the output place of exit, which does not influence any enabledness condition.

7.2.2 User interface integration

Simulation runs can be initiated and managed using the graphical user interface provided by the ViatraDSM framework (Chapter 3). The interface provides support for three key functionalities: (i) interaction with the simulator (scheduling simulation rules to be fired when the simulator enters the IDLE state); (ii) domain-specific model query and manipulation by graph patterns, elementary operations and graph transformation rules, while the simulation is being executed (waiting in the IDLE state); (iii) simulation rule management meaning that simulation rules can be added and removed from the active simulation rule registry any time.

Interaction with the simulator In Figure 7.4, the interactive Petri net simulation is displayed in the concrete syntax model (diagram), shown on the left. In addition to the synchronization of abstract syntax models (tree viewer) and diagrams, the system provides an extensible programming interface to provide decoration for elements included in the context of simulation rules; in Figure 7.4, fireable transitions are highlighted with red colour. In this case, these fireable transitions correspond to various matches of the only simulation rule (Figure 7.1) and the user may select any one to be scheduled for execution ("fired").
7.3 Performance evaluation

Model manipulation and simulation In the IDLE state of the simulator (even when activated simulation rules are available), the user may freely manipulate the model. As these operations are propagated through the RETE network, the corresponding simulation rules are updated and added to or removed from the simulation rule queue automatically. As a result, the user can perform wide scale "debugging" of a simulation run and continuously interact with the system to e.g. try different execution paths without having to restart from scratch.

The ViatraDSM framework allows such editing operations to be carried out through both generic and domain-specific views; as detailed in Chapter 3, domain-specific views offer a syntax-driven editing interface where all editing operations honor the metamodel and ensure the well-formedness of the model. Figure 7.5 shows the tree view-based syntax-driven editor of the ViatraDSM framework; the user may only perform changes allowed by the Petri net metamodel (e.g. only an InArc edge is allowed to be added to the exit transition).

Simulation rule management In addition to the simulation interaction and model manipulation facilities, our simulator also offers to dynamically manage simulation rules: they can be added, changed, and removed without reloading or reinitialising the model space, since the construction of RETE networks is dynamically performed as pattern definitions are loaded. This feature is analogous to hot code replace found in modern program debuggers, and very important for agile development.

7.3 Performance evaluation

From the point of view of performance, the most relevant benchmark for system model simulation is the execution time for simulation sequences as a function of the complexity (size) of models. This shows the scalability of the tool for large models that are typically used for the analysis of complex (or fine-grained) systems. In this section, we overview the performance of our framework for interactive-automated modes using the previously introduced examples.

Concerning execution time, an initial performance benchmark was published for our RETE-based incremental pattern matcher in [22]. The results have shown that the incremental approach in itself
CHAPTER 7. MODEL SIMULATION IN DSMLS

Figure 7.5: Domain-specific syntax driven editing in the ViatraDSM framework

can provide a speed-up of two orders of magnitude (compared to local search-based non-incremental pattern matchers) for as-long-as-possible type model transformations, such as Petri net simulation. This is explained by the fact that the RETE network drastically reduces computation time for iterative pattern matching, which is traditionally the most costly phase.

Since the RETE network essentially sacrifices memory consumption for speed, we have also concluded some initial investigations on how much memory is consumed for the benchmark used in [22]. While the size of the network may vary intensively with the pattern size, the contents of a pattern, and the model sizes, the incremental pattern matching guarantees that as long as there is enough memory, the execution will be fast. All measurements presented in the followings have been carried out on Mac OSX, Java SE 1.6.0 on an Inter Core2 Duo CPU running at 2.4 GHz, with 4GBs of RAM.

7.3.1 Model simulation using Petri nets

Description This scenario summarizes 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. For comparison, we selected the state-of-the-art GrGEN.NET tool [GBG+06] which is widely known to be one of the fastest graph transformation tools available today.

Test case generation In the Petri net test set, we selected regular Petri nets as test cases, which are generated automatically. Here regular means that the number of places and transitions are approximately equal (where their exact ratio is around 1.1). Furthermore, the net has only a low number of tokens, and thus, there are few fireable transitions in each marking.

To generate the elements of the test set, we used six reduction operations (in the inverse direction to increase the size of the net) which are described in [Mur89] as means to preserve safety and liveness properties of the net. These operations are combined with a weighted random operation
7.3. PERFORMANCE EVALUATION

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 2.1 (which is trivially a live net) and the final test graphs are available in PNML [JKW02] format at [fra08]. As the size of a Petri net cannot be described by only a single parameter we used the number of property preserving we applied to indicate the relative size of test cases.

Note that the only assumption we made on our Petri net test cases is to use live and bounded nets to have a potentially unbounded execution sequence. We selected 1000 consecutive transition firings as Short execution sequences and 1000000 transition firings as Long execution sequences. For this benchmark, we compared the total execution time of the simulation sequences.

Characteristics Figure 7.6 presents the feature matrix (according to Section 5.4) describing the Petri net test case.

![Feature matrix of Petri Net benchmark](image)

7.3.1.1 Results

The Petri net synchronization benchmark was executed with short (1000) and long (1000000) 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 7.7: Size of test cases

The size parameters of the nets used as test cases are depicted in Figure 7.7. Net size represents the number of randomly applied inverse property preserving operations used during their generation, while Places, Transitions and Tokens represent their actual number. The results are shown in Figure 7.8 with logarithmically scaled axes, where model size indicates the net size of the test case.

As it can be seen from the graph, our engine (shown as VIATRA/RETE) has a predictable linear scaling up to model size of $10^5$ with a speed of at least two orders of magnitude faster than our engine running with the traditional local search-based implementation of the pattern matching engine (VI-ATRA/LS). 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). Our tool matches and outperforms the GrGEN.NET tool for very large models in case of both short and long execution sequences. Moreover, with additional memory provided, the characteristics of VIATRA2 are expected to better for even larger models with
predictable execution time. Overall, the characteristics show that our implementation is well-suited for interactive applications and scales well even for large models; in practice, simulators built on this engine will allow high performance operation even when using very large models in the modeling environment.

7.4 Related work

While industrial environments such as MetaEdit+ [MEP], Microsoft DSL Tools [Mic], and the Eclipse Graphical Modeling Framework [GMF] provide advanced facilities for the specification of domain-specific languages, they lack high-level support for model execution. For example, in MetaEdit+, simulations can be implemented by hand-coding using an API. Microsoft DSL Tools and GMF both concentrate on a generative approach to ease the development of modeling environments; model execution and transformation in general are yet to be integrated. While OpenArchitectureWare [Ber] can be integrated into GMF editors to provide support for constraint evaluation, currently, there is no support for the utilisation of other features, such as the execution of transformations specified in the xTend Model Transformation language.

There are also several academic DSM frameworks which are complemented with support for model transformations, typically, using a graph transformation [EEKR99] based approach, which approaches show the closest correspondence with our ViatraDSM framework.

The TIGER project [EEHT05] (which is a conceptual continuation of GenGED [BE00] from the 90s) primarily aims at generating syntax-directed editors as Eclipse plugins based upon the EMF and GEF technologies and a graph transformation engine. Recent development [CE06] focused on the tight integration of TIGER’s model transformation infrastructure with Eclipse GMF; TIGER is able to generate GMF-based editors with rich and complex editing facilities (such as the execution of an editing action based on a single graph transformation rule). However, since TIGER lacks a control
flow language, complex simulation steps can only be implemented in Java.

7.5 Summary

In this chapter, we presented a discrete event interactive simulation framework for dynamic domain-specific modeling languages. The dynamic semantics of a language is captured by a event-driven graph transformation rules, and supported by an adapted execution architecture similar to the results of Chapter 5. Our approach is built on the incremental graph pattern matching engine, which instantly identifies all contexts where the enabledness condition of a simulation rule is enabled. As a result, simulations requiring complex model changes (even with intensive creation and deletion of objects) can be executed in an efficient way. We demonstrated the scalability of the simulation engine by comparing it to GrGEN.NET, a state-of-the-art, highly optimized graph transformation tool.

In interactive execution, the key features are the following: (i) debugging support, which means that the modeling engineer may proceed with execution step-by-step, and each step can be revoked (transaction rollbacks); (ii) direct model manipulation during simulation runs allow the modification of the model which is analogous to heap and stack override in traditional debuggers; (iii) simulation rule replacement is supported to that dynamic semantics may be altered rapidly even while a simulation execution is in progress.

As a contribution to stochastic simulation, a new tool called GraTS [14,12] based on our technology was developed in joint work with prof. Reiko Heckel and Paolo Torrini from the University of Leicester. GraTS supports arbitrary distribution functions that may be used that allow the analysis of semi-Markov processes, in addition to traditional Markov processes. More detail on these results are included in Appendix B.

The results of this chapter are formulated as thesis contributions as follows:

2.3 I developed a novel approach for the design-time simulation of visual domain-specific models [23,28,27,19] (Challenge 3) based on simulation rules specified by an enabledness condition (specified by a graph pattern) and an execution step (specified by graph transformation rules), to provide high-level support for debugging the dynamic semantics of executable domain-specific languages within the editing environment itself.

2.4 I proposed efficient implementation techniques to the simulation framework, which are available in the ViatraDSM tool [26] (Chapter 3). I verified the efficiency of these implementation techniques using benchmarks [21,6,17,11].

The original, preliminary version of the DSM simulator in ViatraDSM (which provided the motivation for my work) has been developed by Dávid Vágó [Vág06].
Part III

Applications in tool integration
Contributions of the current chapter  The current chapter proposes a generalization of the event-driven transformation approach, whereby on-the-fly events are captured (serialized) as model artefacts to be processed by asynchronous, change-driven transformations (Progress map 5). These techniques are used to facilitate change propagation between models hosted in different environments, i.e. for incremental code generation (Challenge 5) where changes in a high-level process model (hosted in a modeling environment) are translated to direct model manipulation sequences over a descriptor (deployed to an application server) (Contribution 2).
8.1 Overview

8.1.1 Motivation

Up to now, the design and execution of batch transformations and live transformations were completely separated, i.e. the same transformation problem had to be formulated completely differently. In this section, we bridge the conceptual gap between batch and event-driven transformations by introducing change-driven model transformations.

More specifically, we first define the concept of a change history model to serve as a history-aware log of elementary model changes, which record causal dependency / timeliness between such changes. We show how change history models can be derived incrementally by live transformations during model editing. Then we describe how change history models can be used to incrementally update a model asynchronously (at any desired time) by propagating changes using batch transformations.

The use of change history models in model-to-model transformation scenarios has far-reaching consequences as incremental model transformations can be constructed with minimal knowledge about the current structure of the target model. For instance, transformations can still be implemented when only identifiers and a model manipulation interface are known, but the rest of the actual target model is non-materialized (i.e. does not exist as an in-memory model within the transformation framework). As a result, these concepts can be easily applied in the context of runtime models as well as incremental model-to-code transformation problems.

8.1.2 Application scenario

Example 22 (Incremental code generation and deployment) The example application scenario is based on an actual tool integration environment developed for the SENSORIA and MOGENTES EU research projects. Here high-level workflow models (with control and data flow links, artefact management and role-based access control) are used to define complex development processes which are executed automatically by the JBoss jBPM workflow engine, in a distributed environment consisting of Eclipse client workstations and Rational Jazz tool servers. The process workflows are designed in a domain-specific language, which is automatically mapped to an annotated version of the jPDL execution language of the workflow engine. jPDL is an XML-based language, which is converted to an XML-DOM representation once the process has been deployed to the workflow engine.

A major design goal was to allow the process designer to edit the process model and make changes without the need for re-deployment. To achieve this, we implemented an asynchronous incremental code synchronizing model transformation. This means that (i) while the user is editing the source process model, the changes made are recorded. Then (ii) these changes can be mapped incrementally to the target jPDL XML model without re-generating it from scratch. Additionally, (iii) the changes can be applied directly on the deployed XML-DOM representation through jBPM’s process manipulation DOM programming interface, but, (iv) in order to allow the changes to be applied to the remote workflow server, the actual XML-DOM manipulation is executed on a remote host asynchronously to the operations of the process designer.

Example 23 (Tool integration workflow model) A simple tool integration workflow model is given in Figure 8.1a together with its jPDL XML representation (in Figure 8.1b). Moreover, a metamodel of the source language is given in Figure 8.1c. In case of the target language, an interface is provided to manipulate XML documents (see Figure 8.1d).
8.1. OVERVIEW

8.1.3 Concepts

In this work, we investigate a model synchronization scenario where the goal is to asynchronously propagate changes in the source model $M_A$ to the target model $M_B$. This means, that changes in the source model are not mapped on-the-fly to the target model, but the synchronization may take place at any time. However, it is important to stress that the synchronization is still target incremental, i.e. the target model is not re-generated from scratch, but updated according to the changes in the source model.

Moreover, our target scenario also requires that $M_B$ is not materialized in the model transformation framework, but accessed and manipulated directly through an external interface $IF$ of its native environment. This is a significant difference to traditional model transformation environments, where the system relies on model import and export facilities to connect to modeling and model processing tools in the toolchain.

To create asynchronous incremental transformations, we extend traditional transformations (which take models as inputs and produce models as output) by change-driven transformations which
take model manipulation operations as inputs and/or produce model manipulation operations as output. By this approach, our mappings may be executed without the need of materializing source and target models directly in the transformation system, and may also be executed asynchronously in time.

As we still rely on model transformation technology, operations on models need to be represented in the model space by special trace models which encode the changes of models as model manipulation sequences. We call these models change history models (CHMs in short). These models are generated automatically on-the-fly as the source model changes (see CHM_A in the left part of Figure 8.2) using event-driven transformations.

The actual model transformation between the two languages is then carried out by generating a change history model CHM_B for the target language as a separate transformation (see middle part of Figure 8.2, and also note that traceability information between CHM_A and CHM_B can recorded as inter-model links). It is important to emphasise that CHMs are generated synchronously to the model changes, so that they precisely encode the actual sequence of operations that were performed on the model – this applies to all operations, either executed by the user or, e.g., an automated model transformation.

As change history models represent a trace of model evolution, they may be automatically applied to models (see right part of Figure 8.2). More precisely, we combine a snapshot of the model M_B (representing the initial state) and a change history model CHM_B (representing a sequence of operations applicable starting from the initial state) to create the final snapshot M'_B. In other words, the change history model CHM_B represents an "operational difference" between M'_B and M_B, with the order of operations preserved as they were actually performed on M_B.

8.2 Change history models

Change history models are conceptually derived from the model manipulation operations defined on the host language. These operations may be generic (i.e. corresponding to graph-level concepts such as "create node", "create edge", "change attribute value"), or domain-specific (corresponding to complex operations such as "remove subprocess", "split activity sequence"). In this paper, we discuss the generic solution in detail, however, we also show how our approach can be extended to domain-specific languages.

Change history metamodel The generic change history metamodel for VPM host models is shown in Figure 8.3. CHM fragments are derived from the abstract Operation class, which can be optionally
tagged with a Timestamp attribute for time-based tracing of, e.g., user editing actions. Operations are connected to each other by relations of type *next*, which enables the representation of operation sequences (transactions).

It is important to stress that CHMs do not directly reference their corresponding host models, but use fully qualified name (or unique ID) references. The reason for this is two-fold: (i) by using indirect references, CHMs may point to model elements that are no longer existent (e.g., have been deleted by a consecutive operation), and (ii) CHMs are not required to be materialized in the same model space as the host model (symmetrically, host models are not required to be materialized when processing CHMs). This allows decoupling the actual models from the transformation engine which is a requirement for non-invasive scenarios where target models are indirectly manipulated through an interface.

![Generic change history metamodel](image)

Figure 8.3: Generic change history metamodel

By our approach, change history metamodel elements are either *EntityOperations* or *RelationOperations*. Entity operations use the *parentFQN* reference to define the containment hierarchy context in which the target entity is located before the operation represented by the CHM fragment was executed. Analogously, relation operations use *srcFQN* and *trgFQN* to define source and target endpoints of the target relation element (prior to execution). Note that we omitted inheritance edges from *EntityOperation* and *RelationOperation* in Figure 8.3 for the sake of clarity.

All CHM elements correspond to elementary operations in the VPM model space, in the following categories:

- **creation** (shown on the far left): *CreateEntity* and *CreateRelation* represent operations when an entity or relation has been created (an entity in a given container, a relation between a source and target model element). Both CHM fragments carry information on the *type* (*typeFQN*) of the target element.

- **deletions** (shown on the near left): *DeleteEntity* and *DeleteRelation* correspond to deletions of entities and relations.
• **updates** (shown on the near right): `SetValue` indicates an operation where the `value` field of an entity is overwritten; similarly, `SetName` represents an update in the local name of the target (in this case, as always, `targetFQN` points to the original FQN of the target model element, so this CHM fragment needs to be used carefully).

• **moves** (shown on the far right): `MoveEntity` corresponds to the reparenting of an entity in the VPM containment hierarchy. `SetRelationTarget` and `SetRelationSource` represent retargeting and resourcing operations.

### 8.2.1 Automatic generation of CHMs by event-driven transformations

In this section, we demonstrate the concept and application of change-driven transformations (Figure 8.2) using change history models by the elaboration of the motivating scenario described in Section 8.1.2. First, we demonstrate (in Section 8.2.1) how CHMs can be derived automatically by recording model manipulations using event-driven transformations (Section 5.2). We introduce both generic (metamodel-independent) and domain-specific (metamodel-dependent) techniques to achieve this. Then we discuss (in Section 8.2.2) how model transformations can be designed between two CHMs of different languages. Finally, we describe (in Section 8.3.1) how CHMs can be asynchronously processed to incrementally update a model resided in a model repository or within a third-party tool accessed via an external interface.

First, we demonstrate the automatic generation of change history models for recording modification operations carried out on the host model. Model changes may be observed using various approaches, e.g. by model notification mechanisms such as the EMF notification API, where the model persistence framework provides callback functions for elementary model changes. This approach is limited to recording only basic model manipulation operations, i.e. an appearance of a complex model element (e.g. a graph node with attribute values and type information) requires the processing of a `sequence` of elementary operations (e.g. “create node”, “set value”, “assign type”, etc). If the modification operations may be interleaving (e.g. in a distributed transactional environment, where multiple users may edit the same model), it is difficult to process operation sequences on this low abstraction level.

In contrast, **event-driven transformations** define changes on a higher abstraction level as a new match (or lost match) of a corresponding graph pattern. By this approach, we may construct a complex graph pattern from elementary constraints, and the system will automatically track when a new match is found (or a previously existing one is lost) – thus, model manipulation operations may be detected on a higher abstraction level, making it possible to assign change history models not only to elementary operations, but also to domain-specific ones.

#### 8.2.1.1 Basic patterns

Figure 8.4 shows three basic graph patterns and their ViATRA2 transformation language representations. Pattern `entity_in_parent` encompasses a containment substructure where an entity `E` is matched in a given parent entity `Parent`. A new match for this pattern occurs when any entity is created in the host model (when a new match is detected, concrete references as substitutions for pattern variables `E, Parent` are passed to the transformation engine). Similarly, pattern `relation_source_target` corresponds to a relation `R` with its source `S` and target `T` elements, while pattern `modelelement_type` references any model element with its type. These patterns correspond to basic notions of the VPM (typed graph nodes and edges), and may be combined to create precondition patterns for event-driven transformation rules.
8.2. CHANGE HISTORY MODELS

1. pattern
2. entity_in_parent(E,P)=
3. \{
4. entity(Parent);
5. entity(E) in Parent;
6. \}

2. pattern
3. modelelement_type(ME,T)=
4. \{
5. modelElement(Type);
6. instanceof(ME, Type);
7. \}

3. pattern
4. relation_source_target(R,S,T)=
5. \{
6. modelElement(S);
7. modelElement(T);
8. relation(R,S,T);
9. \}

Figure 8.4: Patterns for identifying relevant model manipulation events

8.2.1.2 Generic derivation rules

Example 24 (Generating CHMs by triggers) On the left, Figure 8.5 shows a sample CHM generation rule for tracking the creation of model elements. A triggered graph transformation rule is defined for a composite disjunctive pattern, which combines cases of new appearances of entities and relations into a single event. Condition clauses (when(create(E)), when(create(R))) are used to distinguish between the cases where an entity or a relation was created. Finally, action sequences (encompassed into seq{} rules after the when-clauses) are used to instruct the V/2 engine to instantiate the change history metamodel, create a CreateEntity or CreateRelation model element and set their references to the newly created host model entity/relation.

Example 25 (Execution sequence) The right side of Figure 8.5 shows an example execution sequence of this rule. The sequence starts with a model consisting only of a top-level container node w0 of type Workflow. In Step 2, the user creates a new Invocation node i0 inside w0. Note that on the VPM level, the creation of i0 actually consists of three operations: (1) create entity, (2) set entity type to Invocation, (3) move entity to its container. However, the live transformation engine triggers the execution of handleCreation() only if the subgraph w0 – i0 is complete. In Step 3, handleCreation() is fired with the match \{Parent = w0, E = i0, Type = Invocation\}, and – as the condition create(E) is satisfied in this case – the appropriate CreateEntity instance ce0 is created.

Domain-specific CHMs Change history models can also be defined on a higher abstraction level, directly applicable to domain-specific modeling languages. In Figure 8.6a, a part of the change history metamodel for manipulating jPDL XML documents is shown. This metamodel uses unique IDs to refer to (non-materialized) model elements (as defined in the jPDL standard); since jPDL documents also follow a strict containment hierarchy, creation operations (as depicted in Figure 8.6a) refer to a parentID in which an element is to be created. In the follow-up examples of our case study, we will make use of CreateJPDLNode and CreateJPDLAttribute to illustrate the usage of this domain-specific change history metamodel.

It is important to note that domain-specific CHMs can be created analogously to generic ones, by using more complex graphs as precondition patterns for events. The domain-specific CHM con-
8.2.2 Model transformations between change history models

Since CHMs are automatically derived as models are modified, they essentially represent a sequence of operations that are valid starting from a given model snapshot (Figure 8.2). As such, they may be used to drive mapping transformations between two modeling languages: such a change-driven transformation takes CHMs of the source model and maps them to CHMs of the target model.

This is a crucially different approach with respect to traditional model transformations in the sense that the mapping takes place between model manipulation operations rather than models, which makes non-invasive transformations possible (where the models are not required to be materialized in the transformation system).
Example 26 (Mapping between CHMs) Figure 8.7 shows an example transformation rule where the creation of an Invocation in the domain-specific workflow language is mapped to the creation of a corresponding jPDL Node and its attribute. In this case, a batch graph transformation rule is used, however, this transformation may also be formulated as a live transformation. The rule looks for a CreateEntity element referencing a node of type Invocation, and maps it to the domain-specific CHMs of the jPDL language. As Invocations are represented by jPDL Nodes with an attribute node, the target CHM will consist of two “create”-type elements, chained together by the Operation.next relation.

The core idea of creating CHM transformations is the appropriate manipulation of reference values pointing to their respective host models (as CHMs only carry information on the type of the operation, the contextual information is stored in their references). In this example, we make use of the fact that both source and target models have a strict containment hierarchy (all elements have parents), which is used to map corresponding elements to each other:

- Based on parentFQN in the source model, we calculate the target parent’s ID parentID as name(CE.parentFQN).
- Similarly, the target jPDL node’s ID targetID is calculated as the concatenation of parentID and name(CE.targetFQN) to place the target node under the target parent.
- Finally, the attribute functionName designates a particular function on a remote interface which is invoked when the workflow engine is interpreting an Invocation workflow node. It is represented by a separate node in the jPDL XML-DOM tree. The targetValue attribute of the additional CreateJPDLAtribute element is derived from the appropriate attribute value of Invocation node in source model (as denoted by the ref(CE.targetFQN) function in the sample code).
CHAPTER 8. CHANGE-DRIVEN TRANSFORMATIONS

156

Figure 8.7: Transformation of change history models

The right side of Figure 8.7 shows a sample execution result of the mapCreate() rule. First, in Step 4, the precondition pattern is matched, and a match is found to the subgraph created in Step 3 of Figure 8.5. Following the successful matching, the action sequence is executed to create the domain-specific CHM nodes cjnl0 (corresponding to a creation of a jPDL Node) and cja0 (creation of a jPDL attribute node). These CHM nodes are chained together by a next relation to be executed in sequence.

Designing change-driven transformations When designing transformations of change history models, it is important to focus on the fact that the transformation will operate on operations rather than models. Consequently, the first step in designing such a transformation is to define the concept of operation – which may be generic (graph-level operations), or domain-specific. This requires a partitioning scheme for the host modeling language, where the partitions correspond to parts whose creation/deletion constitutes an operation which can be represented by a CHM fragment.

It is important to note that the granularity of this partitioning can be determined freely (since it is possible to perform the "aggregation" of operations in, e.g. the transformation between CHMs of source-target host languages); however, we have found that it is useful to define these partitions so that they represent a consistent change (i.e. the results of valid modification steps between two
8.3 Applying CHMs to non-materialized models

8.3.1 Processing change history models

On the macro level, change history models are represented as chains of parametrized elementary model manipulation operations. As such, they can be processed linearly, progressing along the chain until the final element is reached (thus modeling the execution of a transaction). The consumption of a CHM element is an interpretative step with the following actions performed in the context defined by the CHM’s references:

- **creation**: the target entity/relation is created with the correct type assignment; entities are created in the container designated by the parent’s fully qualified name (parentFQN), relations are created between source and target elements referenced by sourceFQN and targetFQN, respectively.

- **moves**: for MoveEntity, the target entity is moved to the container designated by newParentFQN; for SetRelationSource, the source end of the target relation is redirected according to newSourceFQN.

- **updates**: SetName and SetValue are mapped to updates in the name and value attributes. SetRelationTarget is handled similarly to SetRelationSource.

- **deletions**: DeleteEntity and DeleteRelation are interpreted as deletions of their targets (targetFQN).

As Figure 8.2 shows, we apply CHMs to manipulate non-materialized models through an interface. The speciality of this scenario is that instead of working on directly accessible in-memory models, the transformation engine calls interface functions which only allow basic queries (based on ids) and elementary manipulation operations. In this case, CHMs are very useful since they allow incremental updates, as they encode directly applicable operation sequences.

8.3.2 Case study technical details

For the jPDL models of the motivating scenario, we mapped the XML-DOM process model manipulation programming interface to ViATRA2’s native function API, which enables the system to invoke arbitrary Java code from the transformation program. The following native functions are used:

- **getElementById(ID)**: retrieves a jPDL element identified by its unique ID.

- **createElement(parentRef,targetID)**: creates a new jPDL DOM element as a child of its parent (identified by parentRef), with a given unique ID (targetID).

- **addElement(elementRef,DocID)**: adds the element elementRef to the jPDL DOM identified by DocID.

- **setContents(elementRef,text)**: sets the textual content of the given DOM element (elementRef) to text.
Example 27 (Interpreting domain-specific CHMs) In this final case study example, we define an application rule based on domain-specific CHMs for the jPDL XML-DOM model (Figure 8.6a). Figure 8.8 shows the `newCompoundJPDLNode()` rule, which is used to interpret a subsequence of CHM chains for the jPDL domain. More precisely, this rule’s precondition matches the pair of `CreateJPDLNode` and `CreateJPDLAttribute` CHM fragments which correspond to the addition of a new “compound” jPDL node (with a specified function invocation attribute). The rule uses native functions `createElement`, `addElement` to instantiate new jPDL XML elements directly in the deployed process model on the workflow server; `setContent` is used to overwrite the attribute node’s textual content.

```plaintext

grule newCompoundJPDLNode (JPDL_DOM) = {
  precondition (CJN, CJA) = {
    CreateJPDLNode (CJN);
    CreateJPDLAttribute (CJA);
    Operation . next (_,CJN, CJA);
  }
  action {
    \ldots // See contents below
  }
}
```

Figure 8.8: Applying CHMs through the jPDL XML-DOM API

The left side of Figure 8.8 shows the final three steps of our running example. In Step 6, the initial state of the deployed workflow model, the process definition corresponding to `Workflow w0` is still empty. During the rule’s execution, first, the jPDL Node `i0` is created (Step 7), and then in Step 8, the attribute node is added with the appropriate textual content. (Debug calls are used to write debugging output to the Viatra2 console.)

The entire algorithm which applies CHMs follows the linear sequence of operations along the relations with type `Operation.next`; the first operation in a transaction can be determined by looking for a CHM fragment without an incoming `Operation.next` edge.

8.4 Related work

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
8.4. RELATED WORK

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 our 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.

[GHM98] presents a characteristic representative of inconsistency management systems, which records modification histories in the form of (model-based) change description objects just like our approach. In contrast to our solution, [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 paper, 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 our 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.

[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 "Does Exist" and "Should Exist"). 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, our approach is focused on reducing the amount of necessary information 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 modifications intact. This technique again relies on massive amounts of information cached in traceability models, by copying certain parts of the source model intro traceability models.

[DKPF09] present and end-to-end framework and proposes a universal language (Traceability Modeling Language, TML) to capture a wide range of traceability models found in common MDSE applications. It also proposes common requirements shared and supported by our approach. [DMKPF10] applies TML to a state-based traceability management scenario, which is complementary to our event-based approach; it is to be noted that our work overcomes a significant limitations of event-focused techniques pointed out in this paper by providing support for the detection of arbitrarily complex events on the modeling level.

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 our approach, which could generate deltas (for different modeling domains) as inputs for the merging process required in software evolution. However, note that our 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, our solution can be categorized as an operation and intentional change-based approach
as model changes are explicitly expressed as transformations, and they are independent from the versions to which they are applied.

A wide scale of tools are available to support the reverse engineering and visualization of source code into high level artefacts [BBFG08, SFFG10]. For example, the FAMOOS project [DD99] whose aim was to build a framework to support the evolution and re-engineering 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 our 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 our graph patterns in their structure, however, our 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 our solution. Moreover, [NLBK09] lacks live transformation support.

**Calculation of model correspondence and differences** Frameworks such as AMW [FBJ05] 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 our approach. However, in contrast to [CDRP07] and [GKP07], our current
In this chapter, we discussed how model synchronization can be carried out using change-driven model transformations, which rely upon the history of model changes. We presented an approach to automatically (and generically) derive change history models by recording changes in a (source) model using live transformations. Then a change history model of the target language is derived by a second (problem-specific) model transformation. Finally, the target change history model can automatically drive the incremental update of the target model itself even in such a case when only an external model manipulation interface is available for the target model. Our approach was exemplified using an incremental code generation case study.

The results of this chapter are formulated as thesis contributions as follows:

3.1 I developed the concept of change-driven transformations (CDTs) \[15\], which operate on changes (represented as serialized change models or event objects) and weakly referenced host models, in order to allow asynchronous change propagation between non-materialized models (Section 8.1.3).

3.2 I proposed new implementation techniques and a software framework for event-driven and non-intrusive incremental code generation (Challenge 5), where changes are propagated to non-materialized (deployed) models \[4\]. I elaborated a metamodeling framework for creating change history reference models, which represent atomic and complex changes applicable to both materialized and external host models (Section 8.2).
3.5 I applied the incremental code generation approach to a custom domain-specific process description language \cite{15,4} (Section 8.2.1–8.3).
Chapter 9

Tool integration based on change-driven transformations

Progress map 6: Applications of change-driven transformations in tool integration

Contributions of the current chapter This final chapter presents a novel tool integration framework that provides supporting case studies of the language engineering and model transformation-related contributions of the thesis (Progress map 6). The unique aspects of the technology presented in this chapter are: (i) it is built on a systematic conceptualization of different development activity types; (ii) these concepts are synthesized into a modeling language that reuses best practices from established industry standards; (iii) information propagation between components is facilitated by efficient change-driven transformations (Contribution 2). The chapter presents a case study from the domain of embedded software development for aerospace applications.
9.1 The architecture of tool integration chains

9.1.1 Classification of development activities

Complex development processes make use of a multitude of development, design, verification and documentation tools. The wide spectrum of underlying technologies, data representation formats and communication means has called for tool integration frameworks to address the need for a common underlying tool integration platform. Such middleware is typically designed to allow for various tools to be integrated as services, so that the integration process can be designed by concentrating on the tasks that are to be performed, rather than the underlying technological peculiarities.

On the conceptual level, the main functionality of each step (task) is to transform an input artifact into one or more output artifacts. This transformation view on development and tool integration tasks does not have a direct impact on the level of automation. For example, certain tasks can be either (fully) automated, such as compiling source code from an executable model like statecharts or running a model analysis task to reveal conceptual flaws in the design. Other development tasks are inherently user guided (or user driven) where a development step is completed in close interaction with the systems engineers. User guided steps typically include those where design decisions need to be made and recorded, such as modeling. While full automation is impossible (or impractical) for user guided steps, the step itself can still be interpreted using this transformational view. Moreover, automation may still implant design intelligence into such tools be performing on-the-fly validation of certain design constraints, which can reduce costs.

Development steps can also be categorized on the basis of comparing the information between the source and the target formalisms of the step.

- **Synthesis steps** (carried out by using textual or graphical editors, and even certain automated tools like schedulers, optimizers) add new information to the system under design during the completion of the step.

- **Analysis steps** (also known as verification and validation steps), on the contrary, typically abstract from existing information in order to enable checking for certain correctness properties to reveal errors in the design.

- **Derivation steps** (like code generation or model export and import with format conversion) do not add or remove information, however, they change the representation of the information.

<table>
<thead>
<tr>
<th></th>
<th>Automation</th>
<th>Guidance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>Constraint checker, model analyzer</td>
<td>Code reviewer, test case designer</td>
</tr>
<tr>
<td>Synthesis</td>
<td>Scheduler</td>
<td>Modeling tool</td>
</tr>
<tr>
<td>Derivation</td>
<td>Code generator</td>
<td>Guided transformation</td>
</tr>
</tbody>
</table>

Figure 9.1: Process activity types

A summary of these categories are shown in Figure 9.1. It is important to stress that the mode of execution and design information handling aspects are practically orthogonal, so relevant examples for all combinations can be easily given (in the table cells in Figure 9.1).
9.1.2 Synthesis

Synthesis activities are typically carried out by off-the-shelf development tools such as programming environments, documentation tools, modeling editors etc. By these means, engineers create and manipulate the artefacts of the design process using manual or semi-automated procedures.

Along with design information, most (critical and expensive) errors (e.g. design flaws, anti-patterns) are introduced into the system-under-design in these activities. To reduce the impact of these faults, advanced tools offer checking facilities, ranging from light-weight approaches (such as syntax analysis for source code, domain-specific well-formedness checking in model editors) to more advanced features (e.g. static code analysis [BDG+06], continuous software quality monitoring frameworks [BBFG08], and model simulation by in-place execution as discussed in Chapter 7).

The quality of design synthesis output can also be improved by using semi-automated tools for design-time optimization. For instance, in embedded architecture design, automatic schedulers may be used to calculate the timing of a message bus communication protocol, or resource allocation tools may be used to optimally assign software tasks to hardware nodes.

9.1.3 Analysis

Figure 9.2 shows a typical approach to early analysis using model-based techniques. In model-driven development, system requirements and design are captured by high-level, visual engineering models (using popular and standardized modeling languages like UML, SysML, AADL). In critical systems, where the system under design must conform to high quality and reliability standards, early systematic formal analysis of design models plays an increasingly important role to reveal design flaws as early as possible. In general, this can be carried out by generating appropriate mathematical models by automated model transformations. Formal analysis then retrieves a list of problems, which can be fixed by refinement corrections prior to investing in manual coding for implementation. Finally, these refined models may serve as input to code generators and deployment configuration generation, which create the runnable source code of the application as well as parametrize auxiliary deployment infrastructure services such as monitoring.
Analysis steps may include (i) to investigate functional correctness of the design by verifying safety or liveness properties (e.g. by model checking statecharts [PM05b, SPM08]), (ii) to assess the effects of error propagation (e.g. using fault modeling and analysis techniques [Par07]), (iii) to evaluate non-functional characteristics of the system such as performance, availability, reliability or security (e.g. by deriving stochastic models from engineering models [MPB02, SAE06, BPR09]) and many more. A commonality in these approaches is the extensive use of automated model transformations to carry out the abstraction necessitated by the formal analysis.

A practical application of this tool chain architecture, based on ViatRA2 transformations, has been elaborated in [KVG08]. Here, an approach is presented for the verification of business workflows captured by BPEL processes, with full support for compensation and fault handling. BPEL processes are transformed into the SAL language [SAL] to carry out (i) verification of safety and reachability properties and (ii) fault modeling by using the SAL symbolic model checker.

9.1.4 Derivation

Derivation steps primarily include automated code generation tasks, or to chain up several development steps by importing and exporting models in different tools.

Derivation steps can frequently be fully automated as all information required for code generation is available prior to initiating the step. Of course, such code generators may still combine the information embedded in different models to derive the designated output, or to mark the platform independent models by design decisions. Anyhow, in both cases, the actual derivation step is completed using existing information implanted by synthesis steps.

Code generators may derive the source code or the target application (see code generators of statecharts [PM07]), yield deployment descriptors for the target reliable platform [GAV06, KV07], or generate runtime monitors [PM05a].

As a summary, complex tool integration frameworks should be closely aligned with development processes by taking a transformation-based view on individual development steps. Moreover, they need to simultaneously provide support to integrated automated as well as interactive, user-guided development steps where the starting point and the completion of each step needs to be precisely identified. Finally, the framework should enable to integrate arbitrary kind of development steps including synthesis, analysis and derivation tasks.

9.2 A metamodel for development and integration processes

Based on the experience in tool integration summarized in Section 9.1, we propose an integrated approach for designing and executing tool integration processes for model driven development. As a proof-of-concept, we describe a case study developed for the DIANA [Theb] research project (Sections 9.3 and 9.4).

By our approach, development processes are formally captured by workflow models, which (i) specify the temporal macro structure of the process as a task-oriented, hierarchic workflow model; (ii) precisely map the steps of the process to the development infrastructure, consisting of human resources (roles), tools, supporting technologies available as services and development artefacts as entities in the data repository; (iii) define high-level contracts to each step, which specify constraints that help to verify and trace the correctness of outcomes and activities.

These process models are deployed to a software infrastructure, which serves as an automated execution and monitoring platform for the development process. It provides support for running
automated and user-guided activities in a distributed environment consisting of heterogeneous software tools, and a model bus-like [KLN04] data repository.

In this section, we describe the specification language of the workflow models in the tool integration domain in detail.

**Macro structure** For the specification of the temporal macro structure of development processes, we follow the notions and concepts of well-known process description languages such as BPMN or XPDL. As the metamodel in Figure 9.3 shows, processes are constructed from workflow steps (corresponding to distinct activities carried out during development), and may use standard control flow features such as sequences (ProcessNode.next), concurrency (fork-join) and decision points.

![Figure 9.3: Integration process macro structure metamodel](image)

More advanced constructs, such as waiting states are intentionally omitted from this language, since this is intended to be a very high level description, where only the order (precedence or concurrency) of activities is important; for execution, this language is mapped to a lower level jPDL representation which may be customized and augmented with more advanced behavioral properties.

**Hierarchy** It is important to stress the hierarchical nature of the process description: through the `Step.subNodes` relation, workflow steps may be embedded into each other to create a hierarchical breakdown. This allows the process designer to map the "birds-eye-view" structure development processes (such as phases and iterations) to our language; additionally, it supports "drill-up-drill-down"-style navigation through a complicated workflow, which is important to reduce design complexity for large-scale processes.

**Development infrastructure** Besides the causality breakdown, the language features the following notions of the development infrastructure:
• Artefacts represent data structures of the process (e.g. documentation, models, code, generated files, metadata, traceability records).

• Roles correspond to the participants of the process, humans and external entities who are involved in executing the user-guided activities.

• Components are either Tools or Services which are used by the participants or invoked automatically during development, to complete a specific step.

These concepts enable the precise mapping of the development workflow to the actual execution infrastructure.

Mapping to the execution infrastructure As shown in Figure 9.4, our language can be used to precisely specify the execution of the development process. During process modeling, workflow steps are first assigned into the Activity or Invocation categories, depending on the type of interaction (activities are user-guided while invocations are automated). For user-guided tasks, the language adopts the basic role-based assignment method (taken from BPMN), and optionally supports two basic types of relations (responsible and executor) to indicate which group of users may supervise and actually carry out a task.

Activities may make use of Tools, while invocations refer to Services. From the workflow system’s perspective, both tools and services are external software components, which are accessible through interface adaptors. These are represented in the language as Interfaces (Figure 9.4b), which may be connected to artefacts to indicate data flow (input/output). At run-time, both tools and services shall be triggered by the process execution engine, with parameters referring to data repository automatically supplied, so that the user does not have to care about managing data and files.

Contracts In order to guarantee that the result of a step is acceptable and the process can continue, the definition of contracts [Mey92] is a well known paradigm. The idea is to guard both the input and output of a step by specific constraints. Thus, a contract is composed of a precondition and a postcondition. A precondition defines constraints that needs to be fulfilled by the input of the step in order to allow its execution, while the postcondition guarantees that the process can continue only if its constraints are satisfied by the output.

A detailed example of configuration generation in the context of avionics domain (as part of our case study) can be found in Section 9.3.2.

9.3 Case Study: The DIANA toolchain

DIANA (Distributed, equipment-Independent environment for Advanced avioNics Applications [Theb]) is an aeronautical research and development project. It aims at the definition of an advanced avionics platform named AIDA (Architecture for Independent Distributed Avionics), supporting (i) execution of object-oriented applications over virtual machines [Loc06], (ii) high-level publish subscribe based communication, and (iii) the applicability of model driven system development (MDSD) in the avionics development domain.

The DIANA project aims to create an MDSD-based tool chain for the analysis and generation of ARINC653 [ARI] real-time operating system (RTOS) configuration files from high-level specifications. Transforming these high-level models into RTOS-specific configuration artefacts is a complex
task, which needs to bridge a large abstraction gap by integrating various tools. Moreover, critical design decisions are also made at this stage. For this reason, the use of intermediate domain specific models is advantageous to subdivide the process into well-defined steps and precisely define the interactions and interfaces among the tools used.

In order to introduce the DIANA approach Section 9.3.1 focuses on the models and metamodels used through the mapping process, while Section 9.3.2 gives an overview on the actual steps of the workflow.

### 9.3.1 Models

In the DIANA project the aim of the high-level Platform Independent Model (PIM) is to capture the high-level architectural view of the system along with the definition of the underlying implementation platform, while the Platform Specific Model (PSM) focuses on the communication details and service descriptions.
Platform Independent Models  In order to support already existing modeling tools and languages (e.g., Matlab Simulink, SysML, etc.) we use a common architecture description language called Platform Independent Architecture Description Language (PIADL) for architectural details by extracting relevant information from supported common off-the-shelf models. As for capturing the underlying platform (in our case ARINC653) we use a Platform Description model (PD) capable of describing common resource elements.

- PIADL aims to provide a platform independent architectural-level description of event-based and time-triggered embedded systems using message and publish/subscribe based communication between jobs, having roots in the PIM metamodel of the DECOS research project [DEC].
- The Platform Description (model) describes the resource building blocks, which are available in an AIDA Module to assemble the overall resources of an AIDA component. This mainly includes ARINC653 based elements such as modules, partitions, communication channels, etc.
- In the context of the DIANA project we support Matlab Simulink as a source COTS language. We support only a fraction of the language that conforms with the expressiveness of our PIADL to describe the high-level architecture of the system.

Platform Specific Models  The platform specific models are encapsulated in the AIDA Integrated System Model that contains all relevant low-level details of the modelled system. Essentially based on ARINC653, the integrated model provides extensions and exclusions to support the publish/subscribe communication and service based invocations. Its main parts are the following:

- The Interface Control Documentation (ICD) is used to describe data structures and low-level data representation of AIDA systems, interfaces and services to ease integration of the described element with other parts of the system. It supports both high-level (logical) and low-level (decoding) descriptions and was designed to be compatible with the ARINC653 and ARINC825 data and application interface descriptions.
- The objective of the AIDA System Architecture model is to identify, collect and describe the relations among all elements related to the AIDA system. More precisely, the model focuses on the (i) details of the proposed publish/subscribe based communication, (ii) the multi-static configuration of the AIDA middleware and (iii) the detailed inner description of the partitions allocated for the AIDA system.

In order to support traceability – an essential requirement of DO-178B [RTC] certification –, a trace element is saved in the Trace model for all model elements of the PSM created during the mapping process. Such an element saves all PIM model segments that were used for the creation of a PSM model element. Additionally, trace information is also serialized into separate XMI files for each generated configuration file.

9.3.2 Overview of the DIANA System Modeling process

An extract of the defined workflow for the DIANA System modeling process is depicted in Figure 9.5, using a graphical concrete syntax of the process metamodel presented in Figure 9.3 and Figure 9.4.

The process starts with the definition of a complete PIADL model as the task of the System architect (represented by a human symbol). It can be either manually defined using the (i) external
9.3. CASE STUDY: THE DIANA TOOLCHAIN

PIADL editor (depicted by a wrench icon) as part of the PIADL review step or (ii) derived from a Simulink model.

The near one-to-one derivation is supported by the Simulink/PIADL Converter external tool used in the PIADL Review step. It has an input and an output interface figured by a grey and white socket symbol for the Simulink and the PIADL model, respectively. However, as some AIDA specific parameters cannot be directly derived it requires additional clarification from the system architect. For example, a subsystem block in the Simulink model is mapped to a job in the PIADL, but its modular redundancy value (how many instances of the job are required) is not present in the Simulink model.

The complete PIADL is then imported into the PIM/PSM mapping editor responsible for the analysis and definition of configuration tables interface descriptions. This work is done by the Modeling Engineer. Without going into more details it consists of 25 steps organized into the following main categories:

1. **Application allocation**: contains the PIM imports followed by the allocation of application instances to partitions and steps that define additional constraints on the allocation. It relies on the ViATRA2 framework and depicted by an invocation step.

2. **AIDA ICD definition**: steps related to the description of interfaces and services provided and required by applications. These are user driven mapping steps, where PIM types, messages,
topics and services are refined with platform specific information like encoding, default value, etc. It is supported by the PIM/PSM mapping editor.

3. **Communication allocation**: involves steps in the PIM/PSM Mapping editor that carry out the allocation of inter-partition communication channels and the specification of ports residing on each end of these channels.

4. **Artefact generation**: contains steps that carry out the generation of AIDA middleware model, ARINC653 configuration files for the VxWorks real-time OS and the AIDA ICD descriptor.

Additionally, as a cross cutting aspect traceability information - depicted by the Trace model - is saved during the mapping process.

### 9.4 Implementation of the Tool Integration Framework

To support the application of the high-level process modeling language presented in Section 9.2, we have created a prototype implementation for the tool integration framework. This framework provides the software infrastructure on which the case study of Section 9.3 is executed. A main design goal was to integrate our solution to existing off-the-shelf tools that are used in industry practice; thus, both the process modeling infrastructure, as well as the execution environment rely on standard technologies as much as possible.

**Execution architecture**  The execution of the process is facilitated by a service-oriented architecture, based on the jBoss jBPM [Joh04] workflow execution engine and the Rational Jazz platform [IBM], as an integration middleware between tools, services, and the data repository. Building on this software environment, we have implemented a lightweight API that provides essential components, the overall architecture is shown in Figure 9.6.

**Tool Management**  A Tool or Service represents an external executable program that performs one or more tasks during the development. In order to be easily integrated, especially in the case of services, these software components should ideally be programmatically invocable, i.e., have the business functionality exposed to a well-defined interface which is externally accessible (ranging from command line interfaces to library functions or even web services).

**Connectors**  Connectors are the components that provide uniform interface of the tools for the framework. The connector is also responsible for facilitating data flow between the tool and the artefact repository (optionally, support for explicit traceability may also be implemented in the Connector). The Tool Manager together with the Tool Repository serve as a service directory for available tools and services. It relies on the underlying service facilities of the Rational Jazz and OSGi platforms. These components are responsible for the lifecycle management (initialization, disposal) of integrated tools.

**Data Management**  Models and artefacts of the development process are described (on a high abstraction level) in the Process Model. From this, a storage metamodel is generated, which contains dependency references between artefact classes, and includes metadata (such as creation timestamps, ownership and access control flags) as attributes. Traceability information is also handled as storage
9.4. IMPLEMENTATION OF THE TOOL INTEGRATION FRAMEWORK

metamodel classes. The Artefact Manager is responsible for providing access through a service-oriented API (implemented as a tool/service interface) to data records stored in the Data Repository component.

Process Execution The executing processes can be managed and supervised using the Process Execution User Interface. In our prototypical implementation, it provides a control panel where (i) execution of tasks defined in the platform-specific process model can be initiated, (ii) the state of execution (i.e. the current process node, and the process variable values) can be observed, and (iii) versions of artefact instances and their related metadata can be managed.

The Process Execution Engine is responsible for the execution of the steps defined in the Process Model. The process model is mapped to a low-level executable language (jBoss jPDL [Joh04]), which is executed in a customized jBPM instance. The jPDL description contains auxiliary information that is processed by handler plugins, so that the process executor is able to invoke the integrated tools, services, and access the data repository.

Technical details Our prototype implementation is composed of the following software components:

1. As mentioned before, platform-independent Process modeling is carried out with the Eclipse Process Framework Composer [Thec] (ver. 1.5) (which is essentially the same as Rational Method Composer 7.5) that is based on the SPEM [Obj] metamodel. Platform-specific process models are designed with a domain-specific language and editor developed with the ViatraDSM tool (Chapter 3).

2. Model transformations between platform-independent and platform-specific process models, as well as storage and executable process models are implemented as VIATRA2 transformation
plugins.

3. **Tool management** is performed by the Rational Jazz Platform (ver. 0.6). Tool connectors are implemented as Jazz service plugins. For **data management**, we use the Jazz’s data repository [R. 07]. Artefact management services are provided by Jazz Services working on EMF-based storage models, and the Apache Derby relational database management system is used as the underlying Data Repository.

4. Finally, we integrated the JBoss jBPM workflow engine as a Jazz service [Joh04] (ver 3.2) to be used for **process execution** and monitoring. To allow the jBPM engine to invoke tools through connector interfaces, we implemented light-weight action adaptors which serve as plug-ins to the jBPM environment.

### 9.5 Process modeling languages for tool integration

In practice, the domain-specific language of Section 9.2 is not the only means of designing a development process; in fact, several modeling languages may be involved on two levels of abstraction (Figure 9.7).

**High-level process models**  In today’s industrial practice, development processes are frequently captured in process description languages with a focus on methodology-compliance (i.e. enforcing design principles so that the actual development conforms to standard methods such as the Unified Process, or modern agile approaches such as XP or SCRUM). To address this need from a metamodeling perspective, the Software Process Engineering Metamodel (SPEM) [The08b] has been developed by the OMG. Since then, a number of SPEM-based tools have emerged, and IBM Rational’s Method Composer is one of the most well-known of them. Along with its open-source version, the Eclipse Process Framework Composer [Thec] (shown in Figure 9.8), they are based on pattern re-use by allowing to design according to process libraries that incorporate deep knowledge of both standard methodologies (e.g. OpenUP) and also organization-specific customizations.

As EPF’s language includes support for the high level enumeration of roles and artefacts, with lightweight associations (such as responsibility, input-output), a "platform-independent" representation of development processes may be designed. Note that in these models, all activities appear as **tasks** (Figure 9.8), so there is no information present about which elements are used-guided and which are automated.

Thus, this high level model can mapped to our DSML by a VIATRA2 transformation, preserving the macro structure of the process, and importing an enumeration of roles, tools and artefacts. This
domain-specific model has to be augmented manually to precisely specify how activities interact with artefacts, tools, services, and their interfaces.

Storage models Two types of deployment models are generated from the platform-specific model: (i) the Workflow Model contains the description of the tool-chain to be executed in the format (augmented jPDL) that can be executed by the Process Execution Engine, and (ii) the Storage Model, which is the description of the data structure in a format that is needed to configure the Rational Jazz Data Repository.

In Jazz, storage models [Pet06] are EMF/Ecore-compliant metamodels that define an object-oriented database schema in which artefacts can be stored. Inter-class references indicate cross-references between repository elements (for instance, such cross references may be used to determine which document instances need to be manipulated synchronously to maintain consistency).

A sample storage model extract for the DIANA case study is shown in Figure 9.9. Classes, tagged with the Auditable stereotype, are under versioning persistence management, and can store meta-
data as attributes. These attributes can be queried and processed without retrieving the artefacts in their entirety. Note that in this example, we do not record any traceability information between the Simulink model and the rest of artefacts, hence it is shown as a separate auditable entity in the storage metamodel.

Metadata attributes typically include lightweight traceability information (e.g. creation timestamps, creator IDs which may refer to a particular user or the ID of an automated service), and may also incorporate logs and traces (such as, for instance, the `generation_warn_error_log` attribute for the VxWorks configuration document, which contains the warning and error log emitted by the code generator during generation). These auxiliary records, together with a complete `trace model` (also represented as a persistent and versioned artefact) play an important role in achieving end-to-end traceability.

Based on this storage model, a persistence plug-in is generated for the Jazz repository, which allows the tool connectors (as well as external components) to interact with the database on the artefact level. This interface currently only provides basic access functionality (queries and manipulations):

- getter functions for all document types and their metadata (e.g. `getVxWorks_configurations()`, `.getGenerator_config()` etc.), which directly retrieve data records from the artefact database and wrap them into EMF objects;
- getter functions for complex queries involving storage classes with cross references (e.g. `getTrace_model_transitive(String traceModelId)`, which fetches a particular trace model document together with all its referenced document instances);
- manipulation (setter) functions for all document types and their metadata attributes, which take EMF objects as data transfer parameters (e.g. `storeVxWorks_configuration(VxWorksConfigurationModel)`).

9.6 Related work

**Tool integration** The problem of tool integration has already been studied in many different research projects whose relationships to our proposed approach are now surveyed.

The UniForM WorkBench [Ein99] can be considered as one of the earliest attempts for tool integration due to its built-in support for type safe communication between different tools, version and configuration management. Though system models can be interchanged in a type safe manner by the workbench, it cannot be considered as a model-based approach as a whole.

Several technology dependent approaches have already been proposed for tool integration purposes. One valuable representative of this group is R-OSGi [Jan07], which supports the deployment of distributed applications on computers having the OSGi framework installed. Though the underlying OSGi framework has many advanced services, the centralized management (i.e., loading and unloading) of modules is an inconvenient property of R-OSGi. Another representative is the jETI system [Tiz05], which is a result of redesign and Java-based reimplementiation of the Electronic Tool Integration platform, is an approach based on the Eclipse Plugin architecture whose technology dependency has been reduced by its Web Services support. The jABC submodule of the jETI system enhances Java development environments with remote component execution, high-level graphical coordination and dedicated control via formal methods.

The use of workflows for describing the tool integration process, which is a technique also employed in our approach, has been introduced in the bioinformatics domain in [Fla04]. In this paper,
the authors proposed to describe the cooperation of computational tools and data management modules by workflows.

The first form of metamodel-based tool integration appears in [KLN04], which presents two orthogonal design patterns as well. The first pattern suggests the storage of metadata on a server, and the development of a model bus, on which tools can transfer models via a common model interface protocol. The other pattern proposes the use of workflows for describing the tool integration process in the ESML language.

Model transformations in tool integration

In the followings, tool integration solutions with model transformation support are discussed.

In the authors’ experience, VIATRA2, positioned as a dedicated model transformer, has been successfully applied both in scenarios where the abstraction gap (between source and target languages) was relatively small (such as code generation from MDA-style platform-specific models [GÁV06, GDV08, GDV09] [15], or abstract-concrete syntax synchronization in domain-specific languages (Chapter 6)), as well as mappings with strong abstractions (e.g., the generation of mathematical analysis models from design artefacts, for formal analysis purposes).

The IPSEN approach [KNS99] outlined probably the first integration related scenario, where model transformation techniques played a key role. The aim of IPSEN was to construct an integrated software development environment (SDE) tool, which helped capturing both context-free (i.e., syntactic) and context-sensitive (i.e., graph-based) aspects of languages by textual and graphical editors, respectively. The technique of graph transformation has been heavily used for the development of the tool especially for specifying constraints and translations in the context-sensitive domain.

ModelCVS [Eli06] employs (i) semantic technologies in forms of ontologies to partly automate the integration process, and (ii) QVT transformations, which are generated from these ontology descriptions. As distinctive features, ModelCVS uses Subversion for versioning, EMF and MOF-based metamodels for model representation, and a generic workflow ontology for defining processes. In contrast to our approach, ModelCVS prepares adapters for tools and not for models as these latters are stored in a central repository. Additionally, model transformations are used in ModelCVS for the synchronization of models, and not for the definition of the integration process.

From the model transformation point of view, a similar setup can be found in MOFLON [Fel08, Car08]. Transformations are again used for model synchronization, but in this case, they are defined by triple graph grammars. MOFLON operates on JMI and MOF 2.0 based models.

TopCased ("The Open source toolkit for Critical Systems") [Thei] is a software environment primarily dedicated to the realization of critical embedded systems including hardware and/or software. Topcased promotes model-driven engineering and formal methods as key technologies, such as a model bus-based architecture supporting standard modeling technologies such as EMF, AADL, UML-MARTE, and SysML. For model transformations, TopCased uses ATL [A. 06].

The recent EU projects of ModelWare [Thef] and MODELPLEX [Thee] outline techniques that show certain similarity to our approach. ModelWare aimed at defining and developing the complete infrastructure required for large-scale deployment of MDD strategies and validating it in several business domains. It can (i) provide transparent integration across model, tool, platform, machine boundaries; (ii) support the creation of distributed, multi-user tool chains; (iii) handle many metamodels and artefacts; (iv) integrate interactive and non-interactive tools; and (v) use different technologies for communication. ModelWare offers a process modeling framework, and a model bus for exchanging high-level data that are either Java-based or described by Web Services. On the other hand, it lacks model transformation support, which has only been added in its successor MODELPLEX.
CHAPTER 9. TOOL INTEGRATION BASED ON CHANGE-DRIVEN TRANSFORMATIONS

MODELPLEX has a SPEM2 based toolset for supporting the enactment and execution of processes and is integrable with workflow and project management tools as well.

The clear separation of PIMs and PSMs, which specify tool integration processes with different levels of details can only be found in research projects GENESYS [Thed] and DECOS [Thea], which propose a cross-domain architecture for embedded systems, and a model-driven development process for avionics systems, respectively. As distinctive features, GENESYS supports (i) different modeling languages including UML and many of its profiles, (ii) a service-oriented development of subsystems, (iii) both uni- and bidirectional model transformations with manual, semi-automatic, automatic execution.

9.7 Summary

In this chapter, we proposed a tool integration framework, which is centered around a high-level domain-specific language for process modeling to closely align tool integration and development scenarios. With this approach, model-driven development processes can be described precisely, in detail that is sufficient to capture what can and should be automated, but also flexible enough to support user-guided steps as well. Model transformations provided by graph transformation techniques are responsible for fully automating certain steps in the tool chain (like code generation or model analysis tasks).

In addition to the process-driven specification of tool integration chains, we have also presented an execution framework, which was actively used in various research projects. This framework is built to accommodate a wide spectrum of Eclipse-based or external) tools, and automatically execute development processes designed with our modeling language.

The results of this chapter are formulated as thesis contributions as follows:

3.3 I proposed efficient implementation techniques for remote service invocation in the SENSORIA Development Environment (SDE) [3], a tool integration framework developed within the context of the SENSORIA EU FP6 research project (described in detail in Appendix C).

3.4 Based on the SDE, I proposed an extended implementation architecture for a model-based tool integration framework [16,1], where transformations can be used transparently as cooperating services. Information transfer between automated and semi-automated activities is facilitated using change-driven model transformations (Section 9.1–9.4).

3.5 I elaborated a model-based framework for capturing development workflows (Section 9.5), based on standardised modeling languages (SPEM, EPF, and jPDL) (Section 9.5).
Part IV

Conclusions and Appendix
Chapter

10

Conclusion

As a final conclusion, the results presented in the thesis are compared with the main objectives (Section 1.3). Additionally, I report on how these results have been used in practical applications. I also outline some future directions of basic research and applications.

10.1 Fulfillment of objectives

Challenge 1: The integration of domain-specific language engineering with model transformations

The foundations for the research results presented in this work are provided by the ViatraDSM Framework (Chapter 3), a domain-specific language engineering environment built on the ViATRA2 model transformation system. ViatraDSM adapts state-of-the-art metamodeling techniques (separate metamodeling for abstract and concrete syntaxes, and multi-domain integration based on multiple metalevels) with model transformations to provide support for advanced language engineering aspects such as well-formedness constraint evaluation, design-time execution of dynamic semantics and cross-domain integration.

Challenge 4: Event-driven execution for incremental model transformations

The integration of transformations to language engineering motivated the development of a novel execution scheme for model transformations, which is better suited for interactive applications than traditional batch execution. Thus, I developed the concept of event-driven transformations based on incremental graph pattern matching technology (Chapter 5), developed a new specification approach and integrated it into the ViATRA2 transformation language and designed and implemented an efficient execution architecture. I verified the implementation’s performance by systematic benchmarks.

Challenge 2: The complete separation of abstract and concrete syntax representations in DSMLs, based on integrated model transformations

I applied the novel event-driven transformation technology to elaborate a metamodel-driven model synchronization transformation library to facilitate bidirectional correspondence mappings between abstract and concrete syntax representations of a DSML (Chapter 6). Such transformations allow language engineers to completely separate abstract and concrete syntax representations (which is
a major DSML design challenge), in order to be able to use powerful abstractions to increase the usability of such languages.

**Challenge 3: Design-time simulation of dynamic DSMLs based on event-driven transformations**

I also adapted the architecture elaborated for event-driven transformations to support the *design-time execution of dynamic semantics* of domain-specific modeling languages (Chapter 7). Based on this work, in collaborative research, a novel stochastic graph transformation system simulator (Appendix B) has been built, that enables language engineers to perform large-scale simulation experiments using their dynamic DSMLs, e.g. to analyze behavioral aspects of the system-under-design. I verified the performance of the simulation system by systematic benchmarks.

**Challenge 5: Change-driven model transformations in tool integration applications**

As contributions towards the practical applications of my research results, I applied the domain-specific language engineering and event-driven transformation technologies in software development *tool integration* scenarios. First, as a generalization and extension of event-driven transformations, I developed the concepts of *change-driven model transformations* (Chapter 8) that operate on changes of models as input, and applied this approach to incremental code generation over an exogenous infrastructure where changes are propagated directly to deployed models. This, along with a workflow design language, was used for a tool integration framework (Chapter 9) based on the SENSORIA Development Environment (Appendix C) that provides core technology and an implementation environment for model-driven tool integration.

### 10.2 Applications of new results

In this section, I overview the practical applications of the results of my research.

**The event-driven transformation engine of VIATRA2**

The first version of the event-driven transformation engine of VIATRA2 has been developed in cooperation with two MSc students under my supervision, András Ökrös and Gábor Bergmann (the co-supervisors were Dr. Dániel Varró and Dr. Gergely Varró). Their work won the first prize at the Scientific Students’ Association contest (Tudományos Diákköri Konferencia in Hungarian) of the Faculty of Electrical Engineering at the Budapest University of Technology and Economics in 2007, and again a first prize at the nationwide competition in 2009.

The results of this work, described in Chapter 5 are part of the official VIATRA2 Eclipse.org distribution as of Release 3 [VIA]. As such, it has been applied in various projects at our research group, ranging from research prototypes (model-based IT infrastructure monitoring [Szo10b, Szo10a]), tool integration (SENSORIA FP6 and MOGENTES FP7 EU projects [16]), traceability and model change management (in the SecureChange FP7 EU project [4]). This framework also provides the technological foundations to the newest version of ViatraDSM, as well as shares concepts and code with the constraint satisfaction solver engine [HV09] of VIATRA2 (which has been used in the DIANA EU FP6 project [HVS10]).
### 10.3 Future research directions

#### ViatraDSM

The results of Chapter 3 have been implemented in ViatraDSM, an official add-on to the ViATRA2 release. A number of prototype domain-specific modeling languages (such as the tool integration scenario description language for the MOGENTES project, or multiple DSMLs for stochastic simulation at the University of Leicester) have been implemented using this tool.

#### Tool integration in the Sensoria and MOGENTES EU research projects

The Sensoria Development Environment (SDE, Appendix C) has been developed in co-operation with the project partners, especially the team at the Ludwig-Maximilians-Universität München, with Philip Mayer as the leading contributor. Our contributions (remote invocation support) have been developed in cooperation with Ádám Horváth, who was an MSc student under my supervision. Ádám has summarized his work in a Scientific Students’ Association report (TDK), and won the third prize at the Faculty conference in 2008.

The SDE, and its modified and extended versions, as described in Chapter 9, have been used throughout the Sensoria and later MOGENTES EU research projects by academic and industrial partners around Europe. This research has focused on generating configuration and deployment descriptions for web services based on high-level requirement models (Sensoria) and automated test suite generation and test execution in embedded systems (MOGENTES).

#### Model simulation based on stochastic graph transformations

A collaborative research project between our research group and the group of Prof. Dr. Reiko Heckel at the University of Leicester has been started in 2009, as part of the Sensoria project. The goal of this project is to develop a high performance stochastic graph transformation-based simulator, named VIATRA-GraTS. The technology described in Chapter 7 has been used in this tool. GraTS has since been applied to the simulation of peer-to-peer VoIP networks [12], and used in the MSc-PhD education programme at the University of Leicester.

#### EMF-INCEntityQuery

In joint co-operative research targeting broader industrial applications of our technology, the ViATRA2 team has adapted the incremental pattern matcher ([22,21]) to be used on EMF models, a de-facto standard of model-based development tools. EMF-INCEntityQuery [11] provides an efficient application platform for the results of all of my theses.

### 10.3 Future research directions

#### Metamodelling in the large

The metamodelling framework described in Chapter 2 provides not only the foundations for the current generation of ViATRA2 technology, but also outlines the key research direction for the future: for the upcoming ViATRA2 Release 4, we are planning to create a novel metamodelling environment with extended support for multi-domain integration that incorporates the core techniques of Chapter 3 at a more fundamental level. We believe that the most important, currently unaddressed need in modelling applications is the flexible scalability to very large model sizes, where flexibility means multiple
metalevels and dynamic multityping, and large sizes mean models in the range of 100 million+ model elements.

**Ontology-based language engineering**

A recent trend [ROD10] in language engineering emphasizes the re-use of ontology best practices in domain-specific language design. The core idea is that ontologies offer good tools for rapid domain knowledge synthesis, and novel tools such as OntoDSL [WPS09] help to automatically map such ontologies into domain-specific frameworks such as EMF, so that the conceptual bridge between ontology engineering and DSML design can be bridged. While we see this as a promising approach for future DSML engineering research, a number of problems arise due to some inherent differences (e.g. open and closed world assumptions) between metamodeling and ontologies. It is also challenging to map the rich constraint languages of ontologies to OCL and similar constraint description languages used in DSML frameworks. Current research at our group is focused on this problem, and we are working on a framework for the automated mapping of well-formedness constraints (formulated in ontology languages such as SRML) to graph patterns that can be applied directly in DSMLs of ViatranDSM and EMF (through EMF-INCQuery). In the near future, we also plan to support the inverse direction by translating OCL back to graph patterns.

**Transformation engine improvements**

As Chapters 5 and 7 have highlighted, the event-driven execution scheme can be adapted to a wide range of transformation-based applications that significantly extend the batch execution scheme that originally characterized this technology. Thus, in collaborative work, we plan to adapt this technology to the constraint solver engine [HV09]; this re-factoring will also aid the future development of integrated Viatra2 applications as it will provide an easier-to-use programming interface for a wider range of model transformation functionality (than the current Viatra2 API). We are also working on adapting the key ideas of the approach to a wider technological spectrum outside of Viatra2, by making use of state-of-the-art rule-based engines such as JBoss Drools [DRL10].

**Change-driven model transformations**

We believe that the change-driven transformation approach (Chapter 8) is only the first step towards a novel class of specification and implementation framework for model transformations. As highlighted from the work on event-driven transformations, it is a very interesting research question to investigate the relationship between event-driven and batch transformations, how these may interact and how they can be transformed/converted into each other. We think that the change-driven approach is a promising research direction to address these issues, and the next evolutionary step in this direction is already underway in joint work with Gábor Bergmann: in [4], we aim at the uniformization of change-driven transformations by incorporating change representations directly in the high level specification language as change patterns and change-driven rules. Furthermore, we also investigate ways of extending the implementation infrastructure to be able to execute the same rule descriptions in various change processing scenarios (ranging from on-line event stream processing to offline traceability model interpretation). As a proof-of-concept, the novel EMF-INCQuery framework [11] has been applied to traceability management in security architecture modeling, as a contribution to the SecureChange EU FP7 research project.
Appendix

Model transformation source listings

A.1 Abstract-concrete syntax synchronization transformations
/* Marked with grey background in the figure */

pattern mappedNodeFigure(NodeFigureInstance) =
{
  DiagramElement(NodeFigureInstance);
  NodeMappingElement(NodeMappingInstance);
  MappingElement.diagramElement(_, NodeMappingInstance, NodeFigureInstance);
}

// Create abstract syntax nodes for newly created concrete syntax nodes.
@Trigger(sensitivity='rise')
grule linkNodeFigure() =
{
  precondition pattern lhs(/.../) =
  {
    /* concrete syntax (root) - marked yellow in the figure */
    Diagram(CS_CONT);
    NodeFigure(CS_NODE);
    root(_, CS_CONT, CS_NODE);
    /* abstract syntax - marked orange in the figure */
    Hierarchy(AS_CONT);
    relation(REL, AS_CONT, AS_NODE); // connecting relation
    instanceOf(REL, REL_TYPE); // generics
    instanceOf(AS_NODE, AS_TYPE); // generics
    /* mapping model context - marked white in the figure */
    TopMapping(TR_CONT_MAP);
    TopMapping.diagram(_, TR_MAP, CS_CONT);
    TopMapping.hierarchy(_, TR_MAP, AS_CONT);

    // check that the specification metamodel exists
    find existsInMetaModel(AS_TYPE, CS_TYPE, TR_TYPE);
    // check that it is not processed yet
    neg find mappedNodeFigure(CS_NODE);
  }
  or
  {
    /* concrete syntax (subElement) */
    NodeFigure(CS_CONT);
    NodeFigure(CS_NODE);
    subElements(_, CS_CONT, CS_NODE);
    /* abstract syntax */
    Hierarchy.Node(AS_CONT);
    relation(REL, AS_CONT, AS_NODE); // connecting relation
    instanceOf(AS_NODE, AS_TYPE); // generics
    instanceOf(REL, REL_TYPE); // generics
    /* mapping model context */
    NodeMappingElement(TR_CONT_MAP);
    MappingElement.diagramElement(_, TR_CONT_MAP, CS_CONT);
    nodeMapping(_, TR_CONT_MAP, AS_CONT);

    // check that the specification metamodel exists
    find existsInMetaModel(AS_TYPE, CS_TYPE, TR_TYPE);
    // check that it is not processed yet
    neg find mappedNodeFigure(CS_NODE);
  }
}

action
{
  // create abstract syntax model
  new (AS_TYPE(AS_NODE) in AS_CONT); // create node element
  new (REL_TYPE(REL, AS_CONT, AS_NODE)); // create connecting relation
  // create trace model
  new (TR_TYPE(TR_NODE_MAP)); // create trace node
  new (nodeMapping(_, TR_NODE_MAP, AS_NODE)); // connect to abstract syntax node
  new (diagramElement(_, TR_NODE_MAP, CS_CONT)); // connect to concrete syntax node
  new (mappings(_, CS_NODE, TR_NODE_MAP)); // connect the inverse relationship
}
}

Figure A.1: ViATRA2 code for the linkNodeFigure graph trigger
/* Describes a connected abstract syntax model - trace - concrete syntax model tuple */

pattern mappedPlaceFigure (AS_NODE, TR_MAP_NODE, CS_NODE) =
{
  // abstract syntax
  Place(AS_NODE);
  // concrete syntax
  PlaceFigure(CS_NODE);
  // trace model
  PlaceMapping(TR_MAP_NODE);
  nodeMapping(_, TR_MAP_NODE, AS_NODE);
  mappings(_, AS_NODE, TR_MAP_NODE);
  diagramElements(_, TR_MAP_NODE, CS_NODE);
}

asmfunction numberOfTokens / 1;

@Trigger(sensitivity='fall', priority=2)
g-rule synchTokens() =
{
  precondition pattern pre() = {
    find mappedPlaceFigure(CS_NODE, AS_NODE, TR_MAP_NODE);
    find attributeTrace(CS_NODE, CS_ATTR, TR_MAP_NODE)
  }
  OR {
    find mappedPlaceFigure(CS_NODE, AS_NODE, TR_MAP_NODE);
    find tokenCountTrace(AS_NODE, TR_MAP_NODE);
  }
  action
  {
    when (update(value(CS_ATTR))) do seq {
      // attribute value in the
      let Diff = value(CS_ATTR) - value(TR_MAP_NODE) in
      call addOrRemoveTokens(AS_NODE, Diff);
      // update trace model
      setValue(TR_MAP_NODE, value(CS_ATTR));
    }
    when (update(numberOfTokens(AS_NODE))) do seq {
      // number of token instances in the
      let I = Diff in seq
      {
        if (i>0) try choose Tok with find placeToken(Place, Tok)
        do seq {
          // delete Tokens
          if (I == 0) fail;
          delete(Tok);
          update I = I - 1;
        }
        else seq {
          // create Tokens
          if (I == 0) fail;
          new (Token(Tok) in Place);
          new (tokens(_, Place, Tok));
          update I = I + 1;
        }
      }}
    }
  }
}

rule addOrRemoveTokens(Place, Diff) = seq
{
  let I = Diff in seq
  {
    if (i>0) try choose Tok with find placeToken(Place, Tok)
    do seq {
      // delete Tokens
      if (I == 0) fail;
      delete(Tok);
      update I = I - 1;
    }
    else seq {
      // create Tokens
      if (I == 0) fail;
      new (Token(Tok) in Place);
      new (tokens(_, Place, Tok));
      update I = I + 1;
    }
  }}
Appendix

B

GraTS: a tool for stochastic model simulation

This appendix is based on the joint article [14] with Prof. Dr. Reiko Heckel and Dr. Paolo Torrini.

B.1 Introduction

In addition to interactive execution, our model simulation environment offers automatic execution modes in order to allow the behavioural analysis of domain-specific models. In other tools, such automatisms may simply execute simulation rules randomly, or support the mapping of the dynamic semantics of the language to mathematical domains that may model e.g. real-world processes. In our approach, the simulation engine supports the automatic scheduling of simulation rules according to the semantics of stochastic graph transformation systems (SGTS) [Hec05].

In practice, such systems are frequently used for integrated modelling of architectural reconfiguration and non-functional aspects such as performance and reliability. In its simplest form, a SGTS is a graph transformation system (GTS) where each rule is augmented with a rate of an exponential distribution governing the delay of its application, for the purpose to derive continuous-time Markov chains to verify stochastic properties through model checking.

However, this approach has its limitations:

- Model checking with explicit states does not scale well to models with a large state space. Since performance and reliability properties often depend on the behaviour of large populations of entities (network nodes, processes, services, etc.), this is a very significant limitation in engineering practice.

- Also, exponential distributions do not always provide the best abstraction. For example, the time it takes to make a phone call or transmit a message is more likely to follow a normal distribution.

- There are situations where the distributions do not only depend on the rules, but also on the graphs and matches they are applied to. For example, the time it takes to deliver a message may depend on the distance it has to travel, which may be an attribute of the connection.
To counter these limitations, generalised SGTS [KTH09] allow for general distributions dependent on rule - match pairs. Generalised semi-Markov processes provide a semantic model for such systems, supporting stochastic simulation. Rather than model checking, simulations provide a more flexible tradeoff between analysis effort and confidence in the result and so allow to verify soft performance targets in large-scale systems.

Using the foundations of the RETE-based simulation engine, we developed a tool for the stochastic execution of simulation rules that model generalised semi-Markov processes [14]. In the followings, we briefly outline the operation of this execution mode using a distributed networking model that was analyzed in detail in [12].

**Example 28 (A P2P Network Model)** As a test case, we use an example of a SGTS modelling reconfigurations in a P2P network [Hec05]. In this example, stochastic model simulation may be used to analyse, e.g., the probability of the network being fully connected, so that each participant can communicate with every other one.

The simulation rules in Figure B.1 model basic P2P network reconfigurations. Rule **new** adds a new peer, registers it and links it to an existing peer. Rule **kill** deletes a peer with all links attached. Predicate **disconnected** checks if there are two nodes that are not connected by a path of links labelled
The complete source code for Example 28 can be found in [14]. For performance analysis in Section B.4, we consider two families of systems, $SGTS_{random,x}$ and $SGTS_{smart,x}$. The former has rules $\{new, kill, random\}$ and rates $\sigma(new) = \sigma(kill) = 1$ and $\sigma(random) = x$. In the latter, $random$ is replaced by $smart$ with $\sigma(smart) = x$. In both cases $x$ ranges from 1 to 10,000 to test different ratios between basic and redundancy rules.

## B.2 Stochastic execution of simulation schemes

In order to define a general interface between the stochastic control component of the simulation and existing graph transformation tools used for executing rules, we define SGTS for a generic notion of graph transformation.
Definition 35 (Stochastic simulation rule) A stochastic simulation rule $SMSR$ extends model simulation rules by a special function that is defined in terms of an event occurrence Def. 30. Formally, $SMSR = (MSR, F)$, where $F$ assigns to each match $m$ a continuous distribution function such that $F(m)(0) = 0$: $F : m \rightarrow (\mathbb{R} \rightarrow [0, 1])$.

We encode SGTS into generalised semi-Markov schemes (GSMS), a generalisation of Markov chains associated with generalised semi-Markov processes [DK05]. Here, transitions are independent of past states, but unlike Markov chains they may depend on the time spent in the current one, i.e., interevent times may be non-exponentially distributed.

Definition 36 (Generalized semi-Markov simulation scheme) Formally, a Generalized semi-Markov scheme $SIM_{GSM}$ over a model space $MS$ is a simulation scheme (Def. 34) $SIM_{GSM} = (\{S\}, S_0, \{SMSR\}, \text{act})$ where stochastic simulation rules $\{SMSR\}$ are used instead of plain simulation rules.

Figure B.2 illustrates the definition of a stochastic simulation rule in annotated VTCL syntax. Technically, the definitions of distributions $F$ are not specified in VTCL directly, but loaded from an external file and matched to simulation rules by their IDs; these allow to specify the type of distribution (exponential, normal, etc.) as well as its parameters (rate, mean and variance, respectively).

Stochastic simulation execution algorithm The simulation algorithm of generalized semi-Markov simulation schemes is as follows:

1. **Initialisation** — the simulation time $T$ is initialised to 0 and the set of the enabled matches (event occurrences) is computed For each enabled rule-match pair $e = (SMSR, m)$, a scheduling time $t_e$ is computed by a random number generator (RNG) based on the probability distribution assigned to the event. Timed events are collected as a time-ordered queue, similarly to the priority-ordered queue of Figure 5.6.

2. **At each simulation step** upon reaching the IDLE state:
   a) the first element $k = (e(SMSR, m), t)$ is removed from the state list;
   b) the simulation time $T$ is increased to $t$;
   c) the action sequence of $SMSR$ is executed (scheduling);
   d) the new state list $s'$ is computed: (i) first, all the rule-match entries that have been disabled are removed, next, (ii) entries for each newly enabled rule-match pair with time $t = T + d$ are added to the queue, where $d$ is provided by the RNG depending on $F(m)$; finally, (iii) the time-order of the queue is ensured if necessary.

B.3 Prototype implementation

The prototype implementation of the stochastic simulator is implemented as an add-on to the core Viatra2-ViatraDSM system. This component extends the user interface facilities described in Section 7.2.1 by the following features: (i) the user may enter control parameters of stochastic simulation runs through special model elements directly using the model space editor; (ii) after each simulation run, statistic output is generated on the Viatra2 console and can be exported as CSV files for detailed analysis in e.g. Microsoft Excel. A screenshot example of the tool is shown in Figure B.3.
In order to validate the scalability of the stochastic simulation approach, we ran a number of experiments based on the P2P model of Example 28 \cite{Hec05}, by recreating the scenario of \cite{Hec05}. We ran experiments with 10 different models, 5 versions each of using random and smart rules, with rates ranging through $x \in \{1, 10, 100, 1000, 10000\}$. We performed 5 runs each with a simulation time bound of 10s for each experiment – i.e. no run exceeds 10s (of simulated time) regardless of the number of steps.

Figure B.4 gives the output of an experiment, indicating the version of the model (1st column) followed by the percentage of disconnected states encountered, the average number of steps performed per run, the average maximal extension of the network, and the average time taken for each run. These results confirm the inverse dependency observed in \cite{Hec05} between the rate of the smart rule and the probability of being disconnected, whereas for the random rule an increased rate does not lead to any significant change in reliability – as confirmed by the average number of disconnections modulo square of node number (not shown).

As an indicator of performance, the number of simulation steps per second is observed. Due to the dependence on random distributions, the execution times of each of these runs varied in the interval of 1–8 seconds (of real time), so the peak rule execution performance (for smart:100) corresponds to $3561/8 \approx 450$ steps/second. This again supports our previous conclusions: the engine is capable
### APPENDIX B. STOCHASTIC MODEL SIMULATION

<table>
<thead>
<tr>
<th>Model: P2P</th>
<th>Disconnected</th>
<th>Number of steps</th>
<th>Max number of peers</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>random:1</td>
<td>0.46</td>
<td>33</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>random:10</td>
<td>0.62</td>
<td>71</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>random:100</td>
<td>0.55</td>
<td>86</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>random:1000</td>
<td>0.89</td>
<td>284</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>random:10,000</td>
<td>0.46</td>
<td>116</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>smart:1</td>
<td>1.33</td>
<td>18</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>smart:10</td>
<td>0.01</td>
<td>90</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>smart:100</td>
<td>0.00</td>
<td>3561</td>
<td>48</td>
<td>10</td>
</tr>
<tr>
<td>smart:1000</td>
<td>0.00</td>
<td>998</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>smart:10,000</td>
<td>0.00</td>
<td>62</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure B.4: Stochastic simulation benchmark results

of executing simulation steps "instantaneously" and makes even complicated simulations feasible. A more detailed analysis using an extended P2P model (the Skype protocol) is discussed in [12].
Appendix

C

Tool integration in the SENSORIA Development Environment

This appendix is based on the joint book chapter [3] with Philip Mayer.

C.1 Introduction

The success of the Service-Oriented Architecture (SOA) [Erl05] in both industry and research has resulted in a growing need for tool support for developers of services and service-based systems. Specific support for developing SOA systems is beneficial in all phases of the development process, ranging from modeling to runtime, from analysis to implementation.

The SENSORIA project [WBC+09] has provided tools and techniques for many of the tasks developers are faced with during the development of SOA systems. A key first result of SENSORIA in this context is a set of languages for describing SOA systems, like, for example, the UML profile for services (UML4SOA) [MSK08] and accompanying tool support. However, the main consideration in SENSORIA was rigorous engineering of service-oriented systems with a specific focus on formal verification. As our verification and validation methods are often directly based on a formal model, tool support had to be created for allowing developers to use these methods while staying on their chosen level of abstraction – for example, UML. To deal with this issue, SENSORIA has investigated model transformations, a concept taken from the model-driven architecture (MDA) community, to ease the transition between developer-level models of a SOA system and the formal languages required for verification, with the additional benefit of being able to generate executable code as well. Finally, runtime support for services in the form of dynamic discovery mechanisms requires a broker infrastructure and testing tools which should be accessible during development as well.

Altogether, these considerations have led us to develop a tooling platform, the SENSORIA Development Environment (SDE) [MRH08], which integrates the various tools required in the service development process, including modeling, analysis, code generation, and runtime functionality. The SDE:

1. gives an overview of available tools and their area of application,
2. allows developers to use tools in a homogeneous way, re-arranging tool functionality as required, and
3. enables users to stay on a chosen level of abstraction, hiding formal details as much as possible.

In this chapter, we give an in-depth review of the SDE, integrated tools, and ways of using tools in combination for developing and verifying service-oriented software systems. In Section C.2, we give a high-level overview of the SDE. Section C.3 further details design and implementation of our integration platform. In Section C.4, we give an overview of tools integrated into the SDE. Section C.5 shows examples of how tools can be orchestrated to perform in collaboration. Finally, Section C.6 concludes the chapter.

C.2 High-Level Overview

The Sensoria project aims to support developers of service-oriented software systems at various points in the development process. Specific focus is placed on (formal) verification of service artefacts, which includes appropriate modeling support for developers as well as code generation and runtime support. Through various tools, we are thus able to offer functionality which covers the complete model-driven process of service engineering, which is shown in Figure C.1.

After starting with requirements for a SOA-based system, developers advance to the modeling phase. From this phase, various analyzes of the models may be performed, many of them carried out with the help of automated model transformations. Finally, code is generated from the improved models; runtime support is available for executing this code on various platforms. The figure shows the phases which are covered by tools integrated into the SDE – Modeling, Transformation, Analysis, Code Generation, and Runtime. The following functionality is available in each of these phases:

- **Modeling.** Graphical editors for familiar modeling languages such as UML, which allow intuitive modeling at a high abstraction level, and also text- and tree-based editors for formal languages like process calculi.
C.2. HIGH-LEVEL OVERVIEW

- **Model Transformation Functionality, including Code Generation.** Automated model transformations from UML to process calculi and back to bridge the gap between these worlds; also, generation of executable code (for example, Web Service standards like BPEL).

- **Formal Analysis Functionality.** Model checking and numerical solvers for stochastic methods based on process calculi code defined by the user or generated by model transformation.

- **Runtime Functionality.** Integration of runtime platforms, for example BPEL process engines or the Java runtime as well as runtime support for services, for example dynamic service brokering.

The functionality indicated in the previous list is implemented in various tools, some of which have been developed within SENSORIA, some developed outside of the project (for a full list of SENSORIA tools, see Section C.4). The tools are not only developed at different sites, but are also vastly different with regard to user interface, functionality, required computing power, execution platform and programming language. However, all of the tools contribute to the development process and in many cases deliver artefacts which may serve as input to other tools.

The SENSORIA Development Environment (SDE) provides this functionality through a carefully designed, lightweight integration architecture. This is achieved through the following core features:

- **A SOA-based platform.** The SDE itself is based on a Service-Oriented Architecture, allowing easy integration of tools and querying the platform for available functionality. The tools hosted in the SDE are installed and handled as services.

- **A Composition Infrastructure.** As development of services is a highly individual process and may require several steps and iterations, the SDE offers a composition infrastructure which allows developers to automate commonly used workflows as an orchestration of integrated tools.

- **Hidden Formal Methods.** To allow developers to use formal tools without requiring them to understand the underlying formal semantics, the SDE encourages the use of automated model transformations which translate between high-level models and formal specifications.

As with services in a SOA, tool composition in our integration tool is a lightweight one, i.e., the connection between tools is not a priori fixed and adding additional tools requires only minimal change to the integrated tools. Using the tool-as-a-service metaphor, tools are services, each consisting of functions which can be invoked by the user or other services. Contrary to Web services [WCL+05], user interaction is very important for some software development tools. For example, a modeling tool requires a lot of user interaction – ideally, the modeling tool runs on the computer of the user. A model checker, on the other hand, requires a lot of computing power and thus will most likely run on a dedicated server to be accessed remotely with none or only a minimal, generated UI available. Both use cases are supported in the SDE.

By using a SOA-based infrastructure, combining tools into more complex tool chains is straightforward, i.e. possible via dedicated orchestration languages. A typical scenario for tool composition can be found in the analysis and verification of software; for example, model checkers require a certain input format into which most source models first need to be transformed; the same applies to the output. The SDE contains both a textual (JavaScript) and a graphical (UML-based) orchestration language, allowing users to integrate various tools, thereby handling the data flow between these tools.
APPENDIX C. TOOL INTEGRATION IN THE SDE

Figure C.2: SDE architecture

Local Tool

Local Tool

Remote Tool

e.g. modelling
e.g. code generation
e.g. model checker

SDE Platform

Figure C.2: SDE architecture

tools. Having encapsulated the integrating steps, they can be run over and over again for performing the same steps with different input and output data.

Finally, the SDE aims at providing formal verification tools to pragmatic developers. This requires, as indicated above, the use of model transformations to allow developers to stay on their chosen level of abstraction while still enjoying the results available through rigorous verification methods. Through tool chaining and the ability to install verification tools remotely, the SDE enables an MDA-like approach to the analysis of service artefacts.

Figure C.2 shows the architecture of the SDE. As discussed previously, the integration platform hosts a number of tools as services. Through its dedicated orchestration infrastructure, the SDE allows developers to orchestrate tools to be used in combination, which includes using model transformations and a remote invocation functionality for invoking tools hosted on different machines.

The next section will introduce the technical details of the SDE implementation.

C.3 Design and Implementation

The aim of SENSORIA is to support the creation of service-oriented software by augmenting existing development processes and tools. A requirement for the SDE was therefore to integrate with existing tools and platforms for the development of SOA systems. For this reason, the SDE is based on the well-known Eclipse platform [Ecl09b] and its underlying, service-oriented OSGi [OSG08] framework. OSGi is based on so-called bundles, which are components grouping a set of Java classes and metadata providing among other things name, description, version, exported and imported packages of the bundle. A bundle may provide arbitrary services to the platform.

C.3.1 SDE Core and UI

The technical architecture of the SDE is depicted in Figure C.3, which shows the SDE Platform as an OSGi bundle, its dependencies and dependent bundles.

Fundamentally, all tools are integrated as OSGi bundles which offer certain functions for invocation by the platform. As indicated above, the tools integrated into the SDE are vastly different, ranging from user-driven graphical modeling tools to computationally intensive analysis tools with very basic interaction mechanisms. Thus, it is not possible to define a common API for all tools. In the SDE, this problem is solved by using (declarative) OSGi services for each tool. Furthermore, the SDE allows tools to provide their own UI, but also provides a generic invocation mechanism which enables users to invoke arbitrary functions, either directly or through an orchestration. Finally, tool integration requirements should be kept low to ensure integration of as many tools as possible. The
SDE re-uses OSGi and Eclipse technology and declarative service descriptions which are generated from Java annotations for a fast and straightforward integration process.

As can be seen in Figure C.3, the SDE platform and the integrated tools are based on (R-)OSGi only (or, more specifically, the Equinox implementation of OSGi [Ecl09a]). This means that fundamentally, tools must be implemented in Java, although they may wrap native code or remote invocations as they wish. Being only based on OSGi, they can be invoked completely independently from Eclipse. If they additionally choose to provide a UI, this UI is integrated into and based on the Eclipse platform, as is the UI for the SDE platform itself.

Figure C.4 shows a screenshot of the SDE UI. On the left hand side, the tool browser shows installed tools available for invocation and automation. Tools are grouped by category, allowing quick access by application area. Double-clicking a tool in the browser yields more information about the tool and its functionality. This information is shown in the view in the middle: As an example, an integrated tool for qualitative analysis (WS-Engineer) is shown in more detail. Each tool function displayed here can be invoked by clicking the link and providing the parameters. Finally, on the right, the SENSORIA Blackboard is shown, which is a storage area where tools may place arbitrary objects for later use. Finally, at the bottom, the SENSORIA Shell is displayed, which is a live JavaScript execution environment (see section C.3.2).

As an example for a function invocation, clicking on the bpe1ToFSP() function in the WS-Engineer tool yields the following dialogs, where the data for the single parameter (bpe1) can be selected from various sources (Figure C.5).

Finally, the SDE core integrates with R-OSGi [RAR07] to provide the ability to host tools for external invocation, and connect to remote SDE cores. The tools in the tool view in Figure C.4 (left), for example, are listed under the local core. Further (remote) cores may be added as required, and their tools are then listed and used in the same way as described above. Furthermore, the blackboard (right) also distinguishes between the various cores.
C.3.2 Composing Tools

The SDE provides the ability to compose new tools out of existing ones, a process known as orchestration in the SOA world. Creating orchestrations is possible using two mechanisms: A textual,
C.3. DESIGN AND IMPLEMENTATION

JavaScript-based approach, and a graphical, UML-activity-diagram-like workflow approach.

C.3.2.1 Orchestrating with JavaScript

The ability to use tool APIs directly within JavaScript enables developers to create a workflow by simply invoking tool functions and passing data in-between those functions. To enable the newly created workflow to be usable as a tool in its own right, two things are required: Instead of simply creating a workflow, a JavaScript function definition is required which states a function name and parameters. As each tool, function, parameters, and return types may have descriptions and additional meta-data attached, this meta-data must be specified in some way in the JavaScript source files. Both points have been addressed in the SDE. The first is simple; function definitions are already part of the JavaScript specification. The second was solved by employing a JavaDoc-comment-style approach to meta-data specification. Tags like @description are used to convey meta-data information.

As an example, Figure C.6 (left) shows a script for converting UML2 activity diagrams to BPEL, then analyzing them using the WS-Engineer tool, and finally converting the result back to UML2 sequence diagrams showing the error trace. Figure C.6 (right) shows the converted tool inside the SDE tool browser. Scripts created like this can be used on any SDE installation which has the required tools installed. No particular deployment is necessary save copying the script and registering it with the core.

For testing purposes, the SDE also contains a JavaScript live execution environment, the SDE Shell (Figure C.4), where JavaScript commands can be executed without compiling a complete script.

C.3.2.2 Graphical Orchestration

Besides the ability to use JavaScript for orchestration as indicated above, the SDE also contains the ability to orchestrate tools graphically. The syntax used is that of UML2 activity diagrams, where the
APPENDIX C. TOOL INTEGRATION IN THE SDE

main focus is on data flow, i.e. the flow of information from pin to pin. An activity in the diagram represents one function in the tool to be generated which has input pins (parameters) and one output pin (return type). Inside the activity, actions represent function calls to arbitrary (installed) tools. These actions have pins themselves; data flow edges model the data transfer.

As an example, consider the screenshot in Figure C.7, which shows the orchestration introduced in the previous paragraph as a graphical workflow, including the editor which supports it. The function checkActivity(uml) is modeled as an UML2 activity, and each call to a particular function of an installed tool is modeled as an action. On the right-hand side, the toolbar shows all available tools and the functions they provide. Once modeled, an orchestration such as the one above is converted to a Java class, compiled in-memory and installed as a tool in the SDE.

C.3.3 Extending the Platform

The SOA-based architecture of the SDE makes it easy to add new tools – the SDE publishes a core API and an extension point for registering tools. Basically, each tool is an OSGi bundle with some published API and meta-data XML to register the tool with the SDE core. Thus, creating a facade class and registering the class with the SDE extension point enables tool functionality to be immediately available within the SDE, both for manual invocation and automation. Tools within the SDE are
loosely coupled, as they are fundamentally independent from each other and interact through their published service interfaces only. They may, of course, require other tools to be installed for them to work. This is defined in a declarative way through the Equinox extension mechanism and checked by the platform prior to tool installation. The SDE core also contains a set of Java 5 annotations, which enable tool developers to define their tools and functions without writing any XML. As an example, consider Figure C.8: On the left-hand side, a tool interface with SDE annotations is shown; on the right-hand side, the corresponding tool view in the SDE.

The API defined within the integration tool service bundle provides access to all installed tools. A tool may use this API to verify installation of required tools; search for tools based on meta-data, and invoke functionality as needed. Therefore, it serves as a discovery service which moderates between the tools. Once the connection has been made, communication between tools is done directly.

C.4 Integrated Tools

The SDE, including the integrated tools, is available for download at our dedicated tooling website, http://svn.pst.ifi.lmu.de/trac/sct. The website also contains a tutorial for tool integration and videos demonstrating the SDE in action.

This section lists all tools which have been integrated into the SDE platform, sorted by integrated category.

C.4.1 Modeling

**ArgoUML**  
ArgoUML is an open source UML modeling tool which includes support for all standard UML 1.4 diagrams.

http://argouml.tigris.org/
Rational Software Architect  Rational Software Architect is a UML modeling tool which supports UML2.0 profiles and is built on the Eclipse platform.


MagicDraw  MagicDraw is a platform independent UML modeler with profile support for UML2.

http://www.magicdraw.com/

C.4.2 Transformation and Deployment

Hugo/RT  Hugo/RT is a UML model translator for model checking, theorem proving, and code generation: A UML model containing active classes with state machines, collaborations, interactions, and OCL constraints can be translated into the system languages of the real-time model checker UPPAAL, the on-the-fly model checker SPIN, the system language of the theorem prover KIV, and into Java and SystemC code.

http://www.pst.informatik.uni-muenchen.de/projekte/hugo/

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

http://wiki.eclipse.org/VIATRA2

SOA2WSDL-Transformation  The SOA2WSDL transformation, written in VIATRA2, takes high level UML models and produces WSDL (Web Services Description language) output.

http://viatra.inf.mit.bme.hu/

SRMC/UML Bridge  The SRMC/UML bridge offers facilities for meta-model transformation. It translates a subset of UML2 models (Interactions and State Machines) into an SRMC description for performance evaluation. Results are reflected back into the UML model.

http://groups.inf.ed.ac.uk/srmc/

UML2PEPA Transformation  The UML2PEPA transformation, written in VIATRA2, takes high level UML models and produces PEPA models used for analysis in the PEPA/SRMC tool.

http://viatra.inf.mit.bme.hu/

Modes Parser and Browser  The Modes Parser and Browser is a WS-Engineer plug-in to parse and extract broker requirements from UML2 Modes Models.

http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer
C.4.3 Analysis

**LTSA**  
LTSA is a verification tool for concurrent systems. It checks that the specification of a concurrent system satisfies the properties required of its behavior. In addition, LTSA supports specification animation to facilitate interactive exploration of system behavior.

http://www.doc.ic.ac.uk/ltsa/

**WS-Engineer**  
The LTSA WS-Engineer plug-in is an extension to the LTSA Eclipse Plug-in which allows service models to be described by translation of the service process descriptions, and can be used to perform model-based verification of Web service compositions.

http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer/

**SRMC Core**  
SRMC (Sensoria Reference Markovian Calculus) Core provides support for SRMC, an extension to PEPA. It covers steady-state analysis of the underlying Markov chain of SRMC descriptions.

http://groups.inf.ed.ac.uk/srmc/

**MDD4SOA Protocol Analysis**  
The MDD4SOA Protocol Analysis Tool verifies protocol compliance of services modeled in UML4SOA given a protocol state machine. The verification yields a violation graph in case of an error.

http://www.mdd4soa.eu/

**SPIN**  
Spin is a model checker that can be used for the formal verification of distributed software systems.

http://spinroot.com/

**UPPAAL**  
Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).

http://www.uppaal.com/

**CMC / UMC**  
CMC and UMC are model checkers and analyzers for systems defined by interacting UML state charts. Both allow on-the-fly model checking of abstract behavioral properties in the Socl branching-time state-action based, parametric temporal logic.

http://fmt.isti.cnr.it/cmc/, http://fmt.isti.cnr.it/umc/

**LySa tool**  
LySa is a static analyzer for security protocols defined in the LYSA process calculus. The tool provides a LYSA editor and analyzer, the latter of which will verify properties related to secrecy and authentication.

http://www2.imm.dtu.dk/cs_LySa/lysatool/
C.4.4 Deployment and Runtime

MDD4SOA Transformers The MDD4SOA transformers are a set of EMF transformers for converting UML4SOA models into target languages. Supported are BPEL/WSDL, Java, and Jolie.

http://www.mdd4soa.eu/

UML2AXIS Transformation The UML2AXIS transformation, written in VIATRA2, takes high level UML models and produces Web service code based on the Apache Axis Java library.

http://viatra.inf.mit.bme.hu/

Dino Broker The Dino Broker provides dynamic runtime discovery of services which are described in OWL and WSDL documents, thus enabling developers to bind services which correspond to specific criteria.

http://www.cs.ucl.ac.uk/staff/a.mukhija/dino/

C.5 Tool Applications

The tools listed in the previous section can be combined in various ways to achieve different transformations and analyzes. Figure C.9 lists, non-exhaustively, the links between the tools.

As examples, we provide three scenarios with different tools to give some insights into how tools have been chained together within the SENSORIA project. In the following sections, we use four paragraphs to describe each scenario:

- Use Case describes when and why to use a certain tool chain.
- In Tools Involved, we list the tools required to perform the functionality of the scenario.
- Data Flow shows the individual steps to be executed in the tool chain.
- Finally, Results describes the consequences and benefits of the scenario.

The tool chains may be realized manually, i.e. with the user performing one step after another and storing the intermediate objects on disk or on the blackboard, or automatically by employing the JavaScript orchestrator or the graphical orchestration mechanism.

C.5.1 Checking and Deploying Dervice Orcheestrations

C.5.1.1 Use Case

Using a model-driven approach for developing software has been advocated for some time. SENSORIA addresses this area with a customized UML prole for modeling services and service orchestrations. Besides modeling the orchestration implementation itself, a behavioral protocol can help to assess the external behavior of the orchestration and used to verify the actual implementation. Once a service orchestration has been verified, it needs to be transformed to code in target languages like BPEL or Java to deploy it for execution.
C.5. TOOL APPLICATIONS

Figure C.9: Tool chaining in the SDE
C.5.1.2 Tools Involved

This tool chain includes a UML modeler with profile support, like MagicDraw or Rational Software Architect. A protocol analysis tool (part of MDD4SOA) is used to report on protocol violations. Finally, model transformers (also part of MDD4SOA) are used to transform the UML specifications to code in executable languages (for example, BPEL and WSDL) for deployment.

C.5.1.3 Data Flow

The chain starts with the user who employs a UML modeler to design both the orchestration implementation and the service protocol. The resulting diagrams are saved as documents in the XMI format. These files can then be used by the MDD4SOA Protocol Analyzer, which either reports no protocol violations or creates a violation trace. This process is repeated until the process is error-free. Finally, the UML2 models are read by the MDD4SOA Transformers, which generate the appropriate target code, depending on which language has been selected by the user.

C.5.1.4 Results

Chaining tools together in this fashion enables the developer to quickly react to changes in requirements, as the chain can be run automatically whenever a change has occurred, either informing the user of newly introduced problems in the protocol or, if the protocol is valid, with the new implementation in the selected target language.

C.5.2 Qualitative and Quantitative Analysis

C.5.2.1 Use Case

Service-oriented software systems are commonly distributed – they make use of a network to combine various individual software components to work in coordination to reach a higher-level goal. In general, a SOA system contains many different threads of execution, which run in parallel and interact with one another in nontrivial ways. This poses a difficult problem to software designers, as the interaction of such threads needs to be analyzed in order to ensure that no undesirable effects (such as deadlocks) occur. Furthermore, it is not always clear how the system time is spent during runtime. Therefore, mechanical checkers are needed to verify whether a certain implementation is free from conditions such as deadlocks, and secondly for assessing the runtime characteristics of the overall system.

C.5.2.2 Tools Involved

Again, we employ UML modelers like Rational Software Architect or MagicDraw for the modeling of a service-oriented system written in UML. Based on these models, quantitative analysis as well as qualitative analysis is then performed by the SRMC tool and the WS-Engineer tool, respectively. While the former is able to deal with UML directly, the latter requires the BPEL format as input, so we bring in another tool (one of the MDD4SOA transformers) for converting between UML and BPEL.

C.5.2.3 Data Flow

The chain starts with the user who employs a UML modeler to design a model of communicating systems in UML2. The resulting model, in the format of an UML2 XMI file, can be read directly by the
SRMC tool to report on the distribution of time spent in the various states of the process. Using the MDD4SOA transformers, the UML2 model is converted to BPEL to serve as input for WS-Engineer, which is used to verify the required properties (for example, freeness from dead-locks). Finally, the result of the analysis is shown to the user: The quantitative analysis can be directly annotated to the original UML model (or output as graphs), the qualitative analysis – if resulting in an error trace – is shown as Message Sequence Charts (MSCs) or UML2 sequence diagrams.

C.5.2.4 Results

This tool chain provides the user with a “one-click” verification of the model – instead of requiring the user, as is common in many verification tools, to activate a translation of service implementations, feed the translation through a model parser, compile the model, and invoke a verify option on the model checker. All these single steps are handled by the tool chain and the script used to combine the two different analyzers. Thus, checking becomes less of a hassle and will be executed more often, resulting in higher-quality systems.

C.5.3 Modes-Based Dynamic Runtime Discovery

C.5.3.1 Use Case

One of the promises of the Service-Oriented Architecture is the ability to quickly react to changes, for example – on the business level – a change of a business partner, or – on a technical level – network connection problems or server overload. To deal with these problems, the concept of dynamic service discovery and binding has been introduced, which enables developers to specify, on an abstract level, the properties and constraints required of certain services needed by an orchestration. Specification of such properties, the criteria of when to change the service to be used (specified by “modes”), and testing of the resulting runtime behavior are non-trivial issues, and tool support is needed to make such approaches practical.

C.5.3.2 Tools Involved

The main focus of this tool chain lies on testing of dynamic service discovery, hence the most important tool is the Dino Broker used for service discovery. Serving input to Dino is the Modes Parser and Browser Tool which handles translation of modes from the UML2 models. Dino also requires WSDL and OWL documents for service specification which can, in part, be generated by the VIA-TRA2 SOA2WSDL transformation tool. Again, the initial mode specification is done in UML2, for which a UML2 modeler is required.

C.5.3.3 Data Flow

The chain starts with the user who employs a UML modeler to design a model of a SOA system enhanced with mode specifications and the required constraints on services. The Modes Parser and Browser Tool is then used to convert these specifications to input for the Dino Broker. In parallel, the services to be discovered are deployed to the Dino runtime, either from pre-existing OWL/WSDL specifications or from those generated by the SOA2WSDL transformation. Finally, the developer can employ the Dino Broker front-end which is available through the SDE to test-drive the service discovery, and once satisfied, use the generated documents for the final implementation.
C.5.3.4 Results

The ability to generate input for Dino from UML2 and test-driving the discovery right from within the development environment greatly speeds up the process of finding the right mode and constraint specifications. Automation allows writing test cases for the complete process, thus the user may change the specifications at the beginning of the chain and verify the output stemming from an actual discovery run with the Dino Broker, thus saving time and effort in debugging.

C.6 Summary

In this chapter, we have discussed the need for, requirements of, implementation, and usage of a tool integration platform for the development of service-oriented software systems, the SENSORIA Development Environment (SDE). Based on a service-oriented architecture itself, the SDE contains tools for modeling and analyzing service artefacts as well as generating code and supporting services at runtime, allows remote invocation of tool functionality, and enables composition of tools by a textual and graphical orchestration mechanism. Furthermore, we have discussed integrated tools providing support for the development of SOA software, and have outlined how to use tools in combination on top of the SDE.

We believe that thinking of individual development tools as services and including SOA features like self-describing services, remote invocation, and orchestration into our tooling environment greatly extends the applicability of the integrated tools. By including transformation tools, we ensure that using analysis tools is possible without understanding the details of the underlying formal specifications, thus allowing more developers to profit from rigorous verification of their systems.
List of publications

Number of publications: 29
Number of peer-reviewed publications: 24
Approximate number of independent citations: 50

Book chapters (3)


Journal papers (5)


Conferences and workshops (21)

International conferences and workshops (17)


National conferences (4)


University of Technology and Economics, Department of Measurement and Information Systems, 2008.


Other (6)

Technical reports (4)


Scientific Students’ Association Report


Master’s thesis

Bibliography


BIBLIOGRAPHY


