Automated Model Transformations for the Analysis of IT Systems

PhD thesis

Dániel Varró
MSc in Technical Informatics

Supervisor:
Dr. András Pataricza, PhD
associate professor

Budapest, December 2003
Nyilatkozat

Ahúlirott, Varró Dániel, 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 megjelölt.


A dolgozat bírálatai és a védésről készült jegyzőkönyv a későbbiekben a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Dékáni Hivatalában elérhető.
Automated Model Transformations for the Analysis of IT Systems

Dániel Varró
PhD thesis summary

Abstract. When designing critical applications using the UML, the system models are frequently projected into various mathematical domains (such as Petri nets, transition systems, process algebras, etc.) by model transformations to carry out a formal analysis of the system under design. In the current thesis, I introduce a general, visual yet mathematically precise framework for uniformly specifying a large scale of model transformations within and between modeling languages. I also propose automated means to reason about the correctness of transformations, and to generate a transformation program as the implementation from such high-level specifications.

For most computer controlled systems, especially dependable, real-time systems for critical applications, an effective design process requires an early validation of the concepts and architectural choices, without wasting time and resources to assess whether the system fulfills its requirements or needs some re-design. The Unified Modeling Language (UML) provides a standard and easy-to-understand visual way to capture both the requirements and the system model.

However, a standard modeling language does not alone guarantee the correctness of the design. In order to increase the level of confidence that can be put on a system mathematical tools (based on formal methods like Petri nets, dataflow networks, transition systems, process algebras, etc.) are used to assess the most important system parameters (such as functional correctness, timeliness, performability or dependability). Unfortunately, sophisticated verification tools require a thorough knowledge of the underlying mathematics, and therefore special skills are needed for system designers.

In order to bridge the huge abstraction gap between UML and mathematical models, many approaches (starting from the European ESPRIT project “HIDE”) propose to automatically transform high-level UML based system models into low-level mathematical models, and then back-annotate the results of the formal analysis into the original UML model of the system in order to hide the underlying mathematics.

In the current thesis, I propose visual, automated and mathematically precise means to specify, analyze and implement model transformations within and between various modeling languages.

Mathematics of metamodeling. I define a visual, and formally precise metamodeling (VPM) framework that avoids the problems of MOF metamodeling by uniformly handling arbitrary metamodels and models taken from both engineering and mathematical domains. This uniform treatment is achieved by generalizing the traditional inheritance and instantiation relations and formalizing the structure of traditional mathematical definitions.

Specification of model transformations. Based on the paradigm of graph transformation, I elaborate a mathematically precise and visual formalism that simultaneously supports the (i) meta-level definition of an operational semantics to an arbitrary modeling language and (i) the high-level specification of model transformations between such modeling languages.

Automated program generation. In order to support the implementation of model transformations, I propose automated program (and model) generation techniques that automatically synthesize a transformation program from the high-level specification of the transformation which can be executed on an arbitrary model of the modeling language(s).

Formal verification of model transformations. I present an approach for the automated verification of any specific instance model of an arbitrary modeling language (with static structure defined by metamodeling and operational semantics defined by graph transformation systems) using existing model checker tools. I define consistency criteria for model transformations and I propose methods to formally verify the syntactic (proving language containment) and semantic consistency (aiming at property preservations) of such transformations.
Preface

The current thesis discusses the theoretical concepts of VIATRA (you should already note at this very first paragraph that T is not equal to G!), a model transformation framework in a UML environment throughout many pages. But prior to that let me spend just a page or two to cordially thank all those people how gave significant assistance to me in writing this thesis in a rather informal way. For this purpose, I will proceed approximately in an increasing geographical distance as the significance of such assistance cannot be measured properly.

Anyhow, either geographically, or in the order of importance, at the first place, I should acknowledge András Pataricza, who undertook the supervision of my thesis. He proposed me the currently hot topic of model transformations of UML four years ago when it was a warmish rather than a hot topic in the field. Meanwhile, his long-term “intuition” on suggesting my high-level research goals is most amazing for me since all these goals are proved to be achievable (as demonstrated by this thesis). Moreover they really served as a guideline for me to find the lower level research goals and solutions. And naturally, I should also gratefully mention all the vivid discussions, and all the “know-how” I acquired from him that largely helped me “sell my results” on international forums.

I am very much grateful to many of my colleagues at the Department of Measurement and Information Systems (Budapest University of Technology and Economics, BUTE) for the warm and kind research environment. Discussions with István Majzik and Gábor Huszár on various UML and formal methods related issues should be thankfully emphasized. Moreover, I also really acknowledge Zsímond Pap for all his efforts he spent on using earlier versions of the VIATRA tool.

Although being only indirectly related to the current thesis, I would like to say thank you to András Recski, Iván Bach, Katalin Friedl and Judit Csima for all their help and encouragement, since my presentation skills that proved to be useful in conferences have deep origins in the Formal Languages practical courses I was teaching at the Department of Computation and Information Theory (BUTE). I am also grateful to Peter Széredi for introducing to the wonderful world of Prolog. Lectures of Katalin Pásztor on automated theorem proving and Zoltán Csörszeg on type theory I attended at the Eötvös University were also very helpful.

I have written the majority of this thesis during my three-month visit to the research group of Gregor Engels at the University of Paderborn supported financially by the SeagraVis training network. I am grateful to him for supporting my application. I am also indebted to Reiko Heckel for the many fruitful discussions, past and (hopefully) future cooperation, and the nice weekend programs. Research “interactions” with Luciano Baroni (Politecnico Miñno), Jan Hendrik Hausmann, Jochen M. Küster, Stefan Sauer, and Sebastian Thöne, were profitable during that period. However, I would really like to thank everyone in the group for all his or her kindness preventing me to feel alone.

At this point, I should also acknowledge many researchers including at least Paolo Baldan, Andrea Corradini (both at Univ. Pisa), Hartmut Ehrig (TU Berlin), Hans-Jörg Kreowski, Sabine Kuske (both at Univ. Bremen) Andy Schürr (Univ. Darmstadt), Gabi Taentzer (TU Berlin), Hans Vangheluwe
VI  Preface

(Mc Gill University), Andreas Winter (Univ. Koblenz), Albert Zündorf (Univ. Braunschweig) for the fruitful discussions, and/or their encouragement and support.

In 2001, I was invited by John Rushby to spend four months as an international fellow at Stanford Research Institute (SRI International) at the Computer Science Laboratory. I am very much grateful to him, and many of his colleagues: Leonardo da Moura, Sam Owre, Hassen Saüdi, Natarajan Shankar and Ashish Tiwari is just an “underapproximation” of all those I should acknowledge. I learned extremely much about formal verification during this period, and the underlying ideas of several results in the thesis also originate from this time. I am also thankful to my office mate, Jonathan Ford for spending as tourists a nice time in California during the weekends.

Finally, I get to those people for acknowledgment who were, unfortunately, the furthest from me during my visits in foreign countries and whom I missed a lot, i.e., my beloved family: Szílvia (my little wife), Gergely (my brother, and also a co-author of several papers), and my parents Győző, and Mária. Without their love, encouragement and support I would never succeed.

Dániel Varró, December 2003, Budapest, Hungary
# Contents

1 Introduction ................................................................. 1  
   1.1 MDA: Model Driven Architecture ...................................... 1  
       1.1.1 MDA: Vision and reality ........................................ 1  
       1.1.2 Goals of MDA .................................................. 1  
       1.1.3 The MDA architecture ........................................... 2  
       1.1.4 Advantages of MDA .............................................. 3  
       1.1.5 The role of UML in MDA ....................................... 3  
   1.2 Formal Methods in MDA? ............................................... 4  
       1.2.1 Model transformations in MDA ................................ 4  
       1.2.2 Formal methods in system design ............................ 4  
       1.2.3 Thesis objectives ............................................. 5  
   1.3 Main Usage Scenarios of VIATRA ..................................... 6  
       1.3.1 The transformation designer’s view ........................ 6  
       1.3.2 The application designer’s view ............................ 7  
   1.4 An Architectural Overview of VIATRA ............................... 8  
       1.4.1 Metamodeling .................................................. 8  
       1.4.2 XMI: Uniform representation of model ...................... 9  
       1.4.3 Model transformation rules ................................. 10  
       1.4.4 Correctness and completeness of transformations .......... 10  
       1.4.5 Automated program generation .............................. 11  
       1.4.6 The transformation engine .................................. 11  
       1.4.7 Benchmark model transformations .......................... 12  
       1.4.8 Back-annotation of analysis results ....................... 12  
   1.5 The Structure of the Thesis ....................................... 13  

2 VPM: Mathematics of Metamodeling is Metamodeling Mathematics ...... 15  
   2.1 Motivation: Metamodeling and Mathematics ........................ 15  
   2.2 Specifying the Abstract Syntax of Modeling Languages .......... 18  
       2.2.1 Metamodels .................................................. 18  
       2.2.2 (Instance) Models ........................................... 20  
       2.2.3 The four layer MOF metamodeling architecture ............ 22  
   2.3 Problems of MOF Metamodeling .................................... 23  
       2.3.1 Lack of package (metamodel) inheritance ................... 23  
       2.3.2 Lack of association inheritance ............................ 23  
       2.3.3 Structural redundancies in MOF ........................... 24  
       2.3.4 Lack of multiple instantiation ............................. 24  
       2.3.5 Problems with metalevels .................................. 25  
       2.3.6 Metamodeling vs. software engineering .................... 26
VIII Contents

2.4 VPM: Structural Refinement of Metamodels ........................................ 26
  2.4.1 Visual definition of Petri nets .................................................. 26
  2.4.2 Formal semantics of static model refinement .............................. 28
  2.4.3 Formalizing the Petri net metamodel hierarchy .......................... 31
  2.4.4 Pattern refinement ............................................................... 31
  2.4.5 The four-layer MOF architecture in VPM ................................ 32
2.5 Static Consistency Analysis of Metamodels ........................................... 33
  2.5.1 Refinement graphs .............................................................. 33
  2.5.2 The lattice of a refinement graph .......................................... 34
  2.5.3 The lattice of the sets of refinement graphs ............................. 35
  2.5.4 Practical uses of refinement graphs ...................................... 38
2.6 Conclusions ....................................................................................... 41

3 VPM Models as Abstract State Machines ................................................. 43
  3.1 Motivation: Elementary VPM operations on a uniform semantic representation ... 43
  3.2 An Introduction to Abstract State Machines (ASMs) .......................... 45
    3.2.1 Vocabulary and states of ASMs ............................................. 45
    3.2.2 Terms, variable assignment and formulae .................................. 46
    3.2.3 Transition rules, consistent updates, firing of updates .................. 47
    3.2.4 Abstract state machines ....................................................... 51
    3.2.5 An abstraction/refinement technique for abstract state machines .... 51
  3.3 A Meta-Level Algebraic Representation of VPM Models ......................... 52
    3.3.1 Vocabulary and state of VPM .............................................. 52
    3.3.2 Well-formedness constraints for the algebraic representation of VPM ... 53
  3.4 Elementary Manipulations of VPM Models ......................................... 56
    3.4.1 Manipulating VPM models: an overview .................................. 56
    3.4.2 Auxiliary rules for manipulating VPM models ............................ 57
    3.4.3 Operations for model manipulation ....................................... 59
  3.5 A Model-Level Algebraic Representation of VPM Models ....................... 66
    3.5.1 Characteristics of a model-level representation ....................... 66
    3.5.2 Model-level VPM operations ............................................... 67
    3.5.3 A metamodel hierarchy ....................................................... 68
  3.6 Conclusions ....................................................................................... 69

4 Visual Specification of Modeling Languages and Model Transformations ......... 71
  4.1 Specifying Dynamic Behavior: An Overview ...................................... 71
    4.1.1 Dynamic semantics of models and modeling languages .............. 71
    4.1.2 Specification of model transformations ................................... 72
    4.1.3 Problem statement .............................................................. 72
  4.2 Related results .................................................................................. 72
    4.2.1 Relational approaches .......................................................... 73
    4.2.2 Operational approaches ....................................................... 73
    4.2.3 Graph transformation based approaches .................................. 74
    4.2.4 Model transformation tools .................................................. 75
    4.2.5 Own contribution ............................................................... 77
  4.3 An ASM Semantics for Graph Transformation ..................................... 77
    4.3.1 Graph transformation rules ................................................... 78
    4.3.2 Graph pattern matching ....................................................... 80
    4.3.3 Application of a rule: An informal introduction ....................... 83
    4.3.4 ASM semantics for rule application ..................................... 85
5 Automated Model and Program Generation for Transformations
5.1 Automated Implementation of Model Transformations: An Overview
5.1.1 XSLT as model transformation engine
5.1.2 The complexity of graph transformation
5.1.3 Graph pattern matching strategies
5.1.4 Automated code generation approaches
5.2 Automated Program Generation for and by Model Transformations
5.2.1 Prolog representation of model transformation rules
5.2.2 Implementing the control structures in Prolog
5.2.3 Automated program generation by model transformation
5.3 Correctness of the Prolog Encoding
5.3.1 Correctness of the Prolog representation of graph transformation rules
5.3.2 Correctness of the Prolog encoding of control structures
5.4 UML Action Semantics for Model Transformation Systems
5.4.1 Action Semantics for UML: An overview
5.4.2 Action Semantics for model transformation systems: An overview
5.4.3 Encoding the control flow graph
5.4.4 Action Semantics for pattern matching
5.4.5 Manipulating links and objects
5.4.6 Correctness of the encoding
5.5 Conclusions

6 Automated Formal Verification of Visual Modeling Languages by Model Checking
6.1 Towards Model Checking Graph Transformation Systems
6.1.1 Objectives
6.2 Model Checking Transition Systems
6.2.1 Transition systems
6.2.2 SAL: Symbolic Analysis Laboratory
6.2.3 An ASM encoding of transition systems
6.3 From Graph Transformation Systems to Transition Systems
6.3.1 A conceptual overview of the encoding
6.3.2 A naive encoding of graph transformation systems into transition systems
6.3.3 Optimizations in transition systems
6.4 Proof of Operational Equivalence
6.4.1 Correctness and completeness of the naive encoding
6.4.2 Correctness and completeness of the optimized encoding
6.5 Dining Philosophers: A Case Study for Modeling and Verification
6.5.1 Case 1: Graph transformation on the model level
6.5.2 Case 2: Graph transformation on the meta-level
6.5.3 Assessment of verification results

4.3.5 Transformation sequences and concurrency
4.4 Visual Definition of Modeling Languages and Model Transformations
4.4.1 Control structures
4.4.2 Defining modeling languages: dynamic behavior
4.4.3 Static semantics of a language
4.4.4 Model transformation systems
4.5 Rule Refinement
4.6 Conclusions

Contents

5 Automated Model and Program Generation for Transformations
6 Automated Formal Verification of Visual Modeling Languages by Model Checking
6.6 Conclusions and Future Work ........................................ 162

7 Towards Automated Formal Analysis of Model Transformations ......... 165
  7.1 Formal Analysis: Why and What? .................................. 165
      7.1.1 Correctness criteria of transformations .................. 165
  7.2 Syntactic Correctness and Completeness of Transformations .......... 166
      7.2.1 Model correctness ...................................... 167
      7.2.2 Transformation correctness ............................. 168
      7.2.3 Examples on correctness and completeness ............. 169
      7.2.4 Proving correctness by planner algorithms .......... 174
  7.3 Automated Formal Verification of Model Transformations ............ 175
      7.3.1 Conceptual overview ................................... 175
      7.3.2 Metamodel vs. model level verification of model transformations 177
  7.4 Conclusions and Future Work ..................................... 177

8 Benchmarks on Modeling Languages and Model Transformations .......... 179
  8.1 Case Study on Petri Nets ........................................ 179
      8.1.1 A semi-formal introduction to Petri nets ................ 179
      8.1.2 Semantics of Petri nets by model transformation systems 181
      8.1.3 Simulating a Petri net in the Prolog engine .......... 183
      8.1.4 Rule refinement in defining semantic variations of Petri nets 185
  8.2 Extended Hierarchical Automata: Formal Semantics of UML Statecharts 186
      8.2.1 An informal overview of UML Statecharts ................ 186
      8.2.2 The metamodel of UML Statecharts ...................... 189
      8.2.3 Related work on formal semantics for UML statechart .... 191
      8.2.4 Extended Hierarchical Automata ........................ 192
      8.2.5 An operational semantics of Extended Hierarchical Automata 194
      8.2.6 Extended Hierarchical Automata revisited .............. 199
      8.2.7 Correctness of EHA semantics ........................... 203
  8.3 Case Study: Verifying the EHA2PN Model Transformation .............. 204
      8.3.1 Modeling statecharts as Petri nets ..................... 204
      8.3.2 The EHA2PN model transformation ....................... 205
      8.3.3 Verifying the EHA2PN transformation .................... 208
  8.4 Conclusions ..................................................... 211

9 Conclusions .................................................................. 213
  9.1 Fulfillment of Objectives ......................................... 213
  9.2 Utilization of New Results ....................................... 214

A The Proofs of Theorems ............................................... 217
  B.1 Proofs of Chapter 3 ............................................... 217
  B.2 Proofs of Chapter 4 ............................................... 220
  B.3 Proofs of Chapter 5 ............................................... 222
  B.4 Proofs of Chapter 6 ............................................... 225

C SAL Descriptions of the Dining Philosophers Case Study ............... 229
  C.1 SAL Description for the Model-Level Encoding of Dining Philosophers 229
  C.2 SAL Description of the Meta-Level Encoding of Dining Philosophers 229
  C.3 SAL Descriptions for Model-level Encoding of Philosophers' Statecharts 231

References ..................................................................... 233
1

Introduction

1.1 MDA: Model Driven Architecture

1.1.1 MDA: Vision and reality

Computing infrastructures are expanding their reach in every dimension. New platforms and applications must interoperate with legacy systems. Those who architect computer systems, whether for banks or battleships, face daunting technology choices. To protect their investments and maximize flexibility, they buy hardware that implements open interconnection standards like Ethernet and USB, and software that uses open interface standards like CORBA.

But as computers and networks become faster and cheaper, even interconnection standards must evolve. New technologies constantly appear for new application niches. One needs look no further than the recent rise of XML to see how quickly this can happen.

The Object Management Group (OMG) addresses this reality with MDA, the Model Driven Architecture. MDA supports evolving standards in application domains as diverse as enterprise resource planning, air traffic control and human genome research; standards that are tailored to the needs of these diverse organizations, yet need to survive changes in technology and the proliferation of different kinds of middleware. The OMG Model Driven Architecture addresses the complete life cycle of designing, deploying, integrating, and managing applications as well as data using open standards. MDA-based standards enable organizations to integrate whatever they already have in place with whatever they build today and whatever they build tomorrow.

1.1.2 Goals of MDA

The MDA aims at providing a framework for the creation of 20-year lasting application software in such a context where even the interface between the target application and the underlying execution platform is changing. Such situation frequently arises, for instance, when a language or operating system that becomes obsolete, or when a part of the application should be deployed on a legacy system not having resources to support newer software execution platforms. Thus the overall goal is to provide cross-platform compatibility of application software despite any implementation, or platform specific changes (to the hardware platform, the software execution platform, or the application software interface).

In particular, MDA addresses the challenges of today’s highly networked, constantly changing systems environment by providing an architecture that assures:

- **Portability, reusability**, increasing application reuse and reducing the cost and complexity of application development and management, now and into the future.
- **Cross-platform Interoperability**, using rigorous methods to guarantee that standards based on multiple implementation technologies all implement identical business functions.
1 Introduction

- **Platform Independence**, greatly reducing the time, cost and complexity associated with re-targeting applications for different platforms including those yet to be introduced.
- **Domain Specificity**, through domain-specific models that enable rapid implementation of new, industry-specific applications over diverse platforms.
- **Productivity**, by allowing developers, designers and system administrators to use languages and concepts they are comfortable with, while allowing seamless communication and integration across the teams. Moreover, a significant reduce in costs is attained by models of the target application that can be directly tested and simulated.

MDA provides an open, vendor-neutral approach to the challenge of interoperability, building upon and leveraging the value of OMG’s established modeling standards:

- **Unified Modeling Language (UML)** [120], which provides a complete visual modeling framework for analysis, design, implementation, deployment and documentation of applications by the standard object-oriented modeling language;
- **Meta-Object Facility (MOF)** [122], specifying repositories for domain-specific applications and modeling languages by constructing appropriate structural descriptions called metamodels;
- **XML Metadata Interchange (XMI)** [121], a metamodel-specific XML format for interchanging models between CASE tools of different vendors;
- **Common Warehouse Metamodel (CWM)** [116]; serving as a language for database integration in data mining and warehousing.

Platform-independent application descriptions built using these modeling standards can be realized using any major open or proprietary platform, including CORBA, Java, .NET, XMI/XML, Web-based platforms, and future technologies.

1.1.3 The MDA architecture

The designated architecture of MDA is summarized in Fig. 1.1.

![Fig. 1.1. The MDA architecture](image)

MDA recommends starting the design of an application with a **Platform-Independent Model (PIM)** representing business functionality and behavior, undistorted by technology details in the form of a UML model. Here the architect should build a detailed UML model, including pre- and post-conditions of services in the standard **Object Constraint Language (OCL)** [124], and behavioral specification (dynamic semantics) in **Action Semantics (AS)** [123] language. The PIM model of the application encapsulates industrial best practices by applying appropriate design patterns from predefined pattern libraries.

In the next phase, **Platform-Specific Models (PSMs)** containing software architecture (i.e., middleware technologies like CORBA, J2EE, .NET, and future technologies) dependent information...
in the form of additional UML models are generated from the PIM by applying standard mappings in an MDA tool preferably by automatic model transformations.

Finally, in the code generation phase, sophisticated MDA tools automatically generate all or most of the implementation code for the deployment technology selected by the developer. In order to handle legacy systems, reverse engineering tools automate the discovery of models for re-deploying it on a new platform.

1.1.4 Advantages of MDA

As a result, MDA achieves an increased quality, as the majority of software developers are isolated from implementation details, allowing them to focus on a thorough analysis of the application space. Moreover, defect injection (and the resulting rework) is reduced by automating the implementation phase in which most defects are injected: on a typical program, after requirements definition approximately 2/3 of the defects are injected during coding.

Simultaneously, reducing rework by an early validation of executable UML models through (model-level) simulation and testing yields an increased productivity since the increase in a precise modeling span time is less than the decrease in integration and test span time. In addition, software development time is reduced by at least 20% when automating the implementation phase (up to at least 40-60% of physical source code) by code generators.

Furthermore, cross-platform compatibility is attained by raising the traditional code-level reuse of components to the model level: a PIM database may be reused (as is) on any platform for which a mapping is defined (i.e.: a code generator is developed). As a consequence, MDA models are compatible with any hardware platform, any software execution platform, any application software interface, and any implementation language. Thus bridge generation between different platforms is simplified by the common PIM application models, simplifying creation of integrated applications both within and across enterprises.

1.1.5 The role of UML in MDA

The Unified Modeling Language, the de facto standard modeling language of object-oriented design, [143] is undoubtedly the key technology in MDA. UML has already proved to be successful for a wide range of applications ranging from dependable systems (banking, telecommunications, aeronautics, and many other safety critical applications) to embedded real-time systems (systems reactively interacting with their environment) to e-business applications (systems providing business services in the highly distributed environment of the Internet).

The main success of UML relies in the fact that it is a standard (uniformly understood by customers, designers and programmers), and visual notation (with a set of "intuitive" graphical description techniques that are easily understood by both system developers and domain experts).

However, based upon academic and industrial experiences, recent surveys (such as [98]) 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 he or she needs 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.

Recent initiatives for the UML 2.0 RFP (e.g., the 3rd revised proposal for UML 2.0 Superstructure and Infrastructure [162,163] aim at an in-depth evolution of UML into a core kernel language, and an extensible family of distinct languages (diagrams) having their own semantics.
1 Introduction

1.2 Formal Methods in MDA?

1.2.1 Model transformations in MDA

As a further main observation, the MDA framework highly relies on complex model transformations within and between different models and modeling languages. From this point of view, MDA necessitates in general the simultaneous development of models and modeling languages within the same framework. In practice, transformations are necessitated for at least the following purposes:

- **Model transformations within a language** should control the correctness of consecutive refinement steps during the evolution of the static structure of a model to ensure a consistent distributed development of the application. Furthermore, one can define a rule-based operational semantics directly on models by model transformations.
- **Model transformations between different languages** should provide precise means to project the semantic content of a UML diagram into another one, which is indispensable for a consistent global view of the system under design. Additionally, conceptually similar transformations derive PSMs from the global PIM of the system.
- A visual UML diagram (i.e., a sentence of a language in the UML family) should be transformed into its (individually defined) semantic domain, which process is called model interpretation (or denotational semantics).
- **Code generation** from thoroughly analyzed PSMs to target code can also be interpreted as ordinary model transformations that derive a textual representation of the model (either an executable code for simulation, or an XMI description for model interchange). Thus the overall goal should be to provide code generation facilities of a proven quality for a UML design of a proven quality by model transformation methodology.

According to the OMG, the key technology in MDA is the standard and precise UML modeling language. However, one should simultaneously emphasize the crucial role of model transformations within the MDA framework that will (hopefully) provide the missing link for an improved software development process.

Unfortunately, the problem of specifying model transformations in a precise way (from PIM to PSMs or to mathematical domains and from PSMs to application code) is not sufficiently addressed in MDA. Although the high-level of automation introduces additional requirements on assuring the quality of mappings as even a correct conceptual design (PIM) may implant bugs into an application if the automated transformations themselves are erroneous.

1.2.2 Formal methods in system design

Furthermore, automation does not alone guarantee neither the proper choice of underlying architecture nor the elimination of conceptual flaws from the design since wrong design decisions and defects injected in the requirements analysis or design phases are also deployed automatically into the implementation. Thus such an automated platform as MDA undoubtedly requires a precise means to design the target system in order to validate that all requirements are fulfilled. Typically encountered requirements include properties for functional correctness (like safety or security of the target system) and quantitative parameters (like quality of service, reliability, etc.).

Due to the increased complexity of IT systems and increasing customer requirements for quality of service (QoS) and reliability, traditional testing and simulation based validation of the system under design often fails to reveal critical bottlenecks that might prevent the system from guaranteeing the required quality of service. In such cases, formal methods, which are mathematics-based techniques offering a rigorous and effective way to model, design and analyze computer systems, would provide automated means to pinpoint the conceptual flaws in the design (like incomplete or contradicting specifications).
The use of formal verification and validation tools in IT system design is hindered by a gap between practice-oriented CASE tools and sophisticated mathematical tools. On the one hand, system engineers typically lack the proper mathematical skills required for applying formal verification techniques in the software design process. On the other hand, even if a formal analysis is carried out, the consistency of the manually created mathematical model and the original system is not assured. Moreover, the interpretation of analysis results, i.e., the re-projection of the mathematical analysis results into the designated system is problematic. From the engineering point of view, the notion of dependability is a composite one necessitating the analysis of multiple mathematical properties by using different verification tools.

Therefore, formal methods should be integrated into the MDA software development process by "push-button" techniques (i.e., automated verification and validation tools like model checkers that do not require user interaction during execution) that hide the technicalities of the underlying mathematics from the designer. For that reason, UML-based system models can automatically be transformed into different mathematical domains by mathematical model transformations and the results of the formal analysis are automatically back-annotated to the original UML model allowing the designer to fix conceptual bugs in his (or her) well-known environment. As a summary, the source language of a mathematical model transformation is typically UML, while the target language is mathematical modeling language (like Petri nets, finite automata, etc.);

Previous experiences in HIDE

A former ESPRIT project under the acronym HIDE (carried out together with FAU Erlangen, CNUCE Pisa and two industrial partners [24]) has shown the feasibility of a transformation-based, automated, multi-aspect dependability evaluation of UML designs in terms of the expressiveness of the models, and the run-time complexity of the mathematical analysis. However, the semi-formal transformation algorithms designed and implemented for different purposes (e.g. formal verification of functional properties [105] and quantitative analysis of dependability attributes [26, 90]) raised several problems.

- The lack of unique and formal descriptions of the transformation algorithms resulted in handwritten and rather ad hoc implementations (inconvenient for implementing complex transformations).
- Any formal proof of correctness and completeness of these transformation scripts was almost impossible, hence their uncertain quality remained a bottleneck of the entire transformation based verification approach.
- Each model had to be verified individually and manually although the transformation algorithms have similar underlying algorithmic skeletons.

As a conclusion, a general and automated transformation method was missing, which would generate the target models from a well-formed, high-level specification.

1.2.3 Thesis objectives

In the current thesis, I present the theoretical foundations of the VIATRA (VIual Automated model TRAnsformations) framework [49, 176, 188, 190, 191] which provides a transformation-based verification and validation environment for improving the quality of systems designed within the Unified Modeling Language by automatically checking consistency, completeness, and dependability requirements.

Our model transformation approach (which is an integration of different disciplines of artificial intelligence, and computer engineering) is based on formal mathematical background, and provides a general description methodology together with automatically generated transformation code of a proven quality for a large scale of transformations. The framework also contains a research prototype
tool (partially) implementing the methodology presented here, which was applied in several industrial strength projects.

Such a complex automated model transformation system has to fulfill at least the following requirements [188,191]:

- **Requirement 1** The easy-to-understand (visual) and mathematically precise description of source and target modeling languages and models, which is close to existing industrial standards;
- **Requirement 2** A visual but mathematically precise description of model transformation rules clearly indicating the correspondence between the elements of the UML visual programming paradigm (or any source modeling language) and the target mathematical notation (or an arbitrary target modeling language);
- **Requirement 3** An efficient back-annotation of mathematical analysis results aiming to back-project the results of the analysis into the UML design;
- **Requirement 4** An engine for proving semantic correctness and completeness of transformations;
- **Requirement 5** An automatic model generation based upon the visual transformation rules;

Proven correct transformations necessitate a precise underlying mathematical structure for both source models (like UML) and target models (such as Kripke structures, Petri Nets, computational tree logic, etc.). Additionally, model transformation and back-annotation also have to be specified strictly and precisely (Requirements 1, 2, and 3).

On the other hand, a model transformation system should simultaneously be close to industrial standards otherwise its application will be restricted to academic fields. Moreover, as visual specifications (like UML itself) are more expressive for engineers than pure textual notations, a visual transformation description framework is preferable.

The quality of model transformation (Requirement 4 and 5) should be ensured by an automated proof method for correctness and completeness, which step would be followed by an automated program generation phase. The program derived takes a specific UML model as input and generates the language of a particular verification tool as the output. As a result, the quality bottlenecks originating in the former heuristic implementation (manual coding) could be eliminated.

### 1.3 Main Usage Scenarios of VIATRA

The VIATRA framework can be regarded (and therefore used) from two complimentary views, namely, from the transformation designer’s view and the application designer’s view (as depicted in Fig. 1.2 and 1.3, respectively).

#### 1.3.1 The transformation designer's view

The overall goal of a transformation designer using the VIATRA framework is to develop syntactically and semantically correct model transformations typically from UML as the source language into an arbitrarily chosen target language including mathematical domains (like Petri nets, finite automata, etc.) for formal analysis purposes and traditional programming languages for code generation purposes.

As the computational complexity of transformations into formal analysis tools are typically much higher than that of transformations required for code generation (e.g., in case of UML statecharts, the partial traversal of the state space is required by flattening the hierarchical UML state machines), in the current thesis, we typically demonstrate the feasibility of our concepts on mathematical model transformations.

- **Definition of modeling languages.** Both the UML dialect to be used by the UML modeler of the target design and the input notation of the target mathematical analysis tool are defined by their
respective metamodels. This metamodel based approach offers a high degree of flexibility both towards the UML dialect used (like the inclusion of different OMG standard profiles or adaptation to a modeling and implementation style used in an enterprise) and the target mathematical language.

- **Definition of model transformations.** Transformations can be defined by a set of simple transformation rules relating individual UML conceptual elements to their equivalent target mathematical notation. These transformation rules themselves can be designed in UML thus providing an easy-to-use interface for the designer of the transformation. Meanwhile, their precise formal background is provided by widely used principle of **graph transformation** [45] (a generalization of the well-known Chomsky-grammars used in compiler construction).

- **Automated generation of transformers.** The implementation of transformations (called transformers in the sequel) is automatically derived from the rules in the form of a Prolog program or standard Action Semantics expressions by the automated model and program generation facilities of VIATRA.

- **Verification of transformations.** The transformation designer can guarantee the correctness of his transformations by testing it on simple source models and by automated verification features of VIATRA built on off-the-shelf model checking tools. In this respect, he can prove mathematically, that a model transformation preserves certain semantic properties (like reachability, safety, etc.) in a highly automated way.

### 1.3.2 The application designer’s view

The overall goal of an **application designer** (or systems engineer) is to develop syntactically and semantically consistent UML-based system models of the target application fulfilling the requirements of the customer. For this purpose, he uses the transformers defined by the transformation designer and generated by VIATRA as push-button techniques to project his UML model into various mathematical analysis tools, and to back-annotate the results of the mathematical analysis into the original UML model.

- **Enrichment of the UML model.** The UML-based system model of the target design should be enriched first by local dependability attributes (like fault and repair rates) associated to the individual components. An expert can estimate these attributes in a very similar way as the parameterization of traditional dependability models;

---

Fig. 1.2. Using VIATRA: the transformation designer's view
1 Introduction

- **Capturing requirements.** Qualitative requirements have to be included in the form of a checking automaton\(^1\).
- **Automated generation of mathematical models.** *Automatic derivation of mathematical models* (like hierarchical automata and temporal logic for the proof of logic correctness, Stochastic Petri Nets for the quantitative evaluation of dependability, or queuing networks for performance analysis) from the UML model of the target design under evaluation.
- **Back-annotation of analysis results.** A back-annotation engine (based directly upon the automatically generated transformer) provides the user with the analysis results integrated into the original UML model thus avoiding the interpretation of mathematical results. Note that the entire transformation definition and implementation framework is hidden from the end user in the VIATRA framework, as he deals only with the back-annotated results.
- **Pilot transformations.** Pilot transformations in the VIATRA framework include (i) formal proof of logic correctness of UML statechart specifications, (ii) check of their completeness and consistency, (iii) quantitative dependability evaluations and (iv) high-level fault modeling and analysis.

1.4 An Architectural Overview of VIATRA

The current section gives a brief technological overview of the VIATRA framework by describing a typical scenario of developing model transformations. The process of model transformation is characterized by a model analysis round-trip (illustrated by the sequence of rounded grey boxes in Fig. 1.4).

1.4.1 Metamodeling

A model transformation framework necessitates, on one hand, a *uniform and precise description of source and target models and modeling languages* to improve the quality of such transformations. But, on the other hand, it should *follow the main standards of the industry* as much as possible in order to be integrated to software design methodologies.

For this reason, the metamodeling foundations of VIATRA [178, 191] are primarily based on the **Meta Object Facility (MOF)** metamodeling standard. MOF *metamodels provide graphical means to define modeling languages in various domains* by combining (i) the expressive power of *UML class diagrams* for defining the abstract syntax of the language with the **Object Constraint Language**

\(^1\) Requirement modeling is still one of the least established fields in UML based analysis. Alternate approaches use OCL [67] or some form of temporal logic formulae for requirements modeling.
(OCL) [124] for describing static well-formedness constraints. MOF metamodels are used uniformly to describe UML models (following the standard metamodel of UML) as well as mathematical structures (by creating non-standard metamodels for them).

Unfortunately, our experiments have also revealed that MOF metamodels are insufficient to provide a precise underlying basis for reasoning about semantic properties of model transformations. Therefore, we developed a Visual and Precise Metamodeling (VPM) framework [181, 186] (discussed in Chapter 2) to overcome the problems of MOF concerning, especially, its imprecise semantics and the lack of proper abstraction and refinement mechanisms by generalizing the structures of traditional mathematical definitions.

**Filtering of UML models**

A typical UML model contains more details than required for a specific mathematical analysis (for instance, documentation or use case diagrams are frequently of little importance). Thus, in the sequel, we assume that a UML model will only contain the relevant pieces of information with respect to a specific analysis, and this reduced model can be obtained from the original user-created system model by some filtering mechanism.

In VIATRA, filtering is expressed by metamodels. Exactly those constructs are regarded as relevant (thus transformable) that are included in the metamodel of the source language (hence if specific constructs are irrelevant for one purpose, they are simply omitted from the metamodel).

### 1.4.2 XMI: Uniform representation of model

The front-end and back-end of transformations (UML as the source model and a formal verification tool as the target model) is defined by a uniform, standardized description language of system modeling, that is, **XMI (XML Metadata Interchange)**. XMI is a special metamodel dependent collection of XML constructs providing an XML representation for arbitrary (MOF based) models.

XMI seems to be a natural choice as a large number of UML tool vendors provide a facility to export their models into XMI, moreover, several academic communities (e.g. the Petri Net [3], or the
graph transformation community [157]) have started discussion to settle on a general XML based interchange format for their tools. In the latter case, the upcoming standard is largely influenced by the metamodeling and XMI-based Budapest proposal [180, 189]. As a result of this uniform XMI representation of models, an open, tool-independent architecture is obtained.

1.4.3 Model transformation rules

The visual specification of model transformations is provided by graph transformation [8, 58, 142], which combines the advantages of graphs and rules into an efficient computational paradigm.

A graph transformation rule is a special pair of pattern graphs where the instance defined by the left hand side is substituted with the instance defined by the right hand side when applying such a rule (similarly to the well-known grammar rules of Chomsky in computational linguistics).

Graph transformation rules appear as a formal background for many different kinds of transformations. When defining a visual modeling language (like UML, Petri nets, dataflow networks) metamodeling techniques only provide a means to capture the static structure of well-formed models. However, a precise specification of a modeling language has to include descriptions of the dynamic behavior (or run-time evolution) of models. In this sense, using graph transformation rules a visual but formal technique is attained to capture the dynamic operational semantics of arbitrary modeling languages.

On the other hand, for transformations between different modeling languages, we restricted ourselves to use graph transformation rules of a special structure (denoted as model transformation rules, in the sequel) which provides a closer fit to typical transformation designers' needs.

Transformation rules are specified by using a visual notation of UML. However, for obtaining a tool-independent transformation specification, the transformation rules will also be exported in an XML based format, conforming to the evolving standard of graph transformation systems [157] in a later phase.

1.4.4 Correctness and completeness of transformations

Semi-formal UML descriptions are transformed into different mathematical notations (Petri Nets, temporal logic, etc.) in order to investigate them by a formal and rigorous analysis. Thus the correctness of a UML model is verified in the target language. In this respect, why should one verify the correctness of model transformations?

The main reason is that the results of a formal analysis in a target language are only faithful if there is a close semantic correspondence between the UML and mathematical models. Such a consistency can be obtained if semantic equivalence could be proved by prescribing special requirements that must be fulfilled in both models.

Since a majority of model transformations is a property preserving projection (i.e., some aspects of the source model may not be transformed or having identical images in the target language) thus proving semantic equivalence of the two models may frequently be (also theoretically) impossible. As an alternate solution, invariant properties can be prescribed for the transformation itself (e.g. two connected states in a statechart must be “connected” in a definite way in a corresponding Petri Net or automaton description).

Therefore, the consistency of transformations can be investigated from the following aspects:

- **Syntactic correctness**: In this case, a successful verification step proves that the output target model yielded by the transformation is a well-formed model of the target language.
- **Semantic correctness**: In order to ensure semantic correctness, property preservation (with respect to a set of requirements like reachability, deadlock freedom or safety) has to be proved between UML models and the specific target formal verification language.
• **Completeness**: As an additional problem, the transformation designer also needs to investigate that each construct in the source language is handled by a corresponding rule of the transformations (i.e., he did not forget about anything).

In the VIATRA framework, *syntactic correctness and completeness* can be verified by planner algorithms [187,188,191], while the more interesting *semantic correctness* problem is tackled by projecting model transformation rules into transition systems which provides access to automated *model checking* facilities [146,170–172].

### 1.4.5 Automated program generation

Even if the description of the transformation is theoretically correct and complete, additionally, the source and target models are also mathematically precise, the implementation of these transformations has a high impact (thus a high risk) on the overall quality of a transformation system. Previous experiments (in project HIDE [24]) demonstrated that the quality of an automatically generated executable transformation program is much higher than a manually written target program.

The automated program generation of VIATRA [167,178,185] allows the transformation designers to focus on the design of a model transformation rather than the implementation. Moreover, once the automated program generator is completed, the time and workload related to the design of a single transformation is drastically decreased. As being a logic programming language based on powerful unification methods, Prolog was a suitable choice for a prototype implementation for initial experiments.

VIATRA proposes a *reflective* method for the automatic generation of transformers derived from the high-level specification consisting of graph transformation rules. The program generator takes a UML profile tailored to model transformation systems as the input, and produces the output Prolog program by successive model transformation steps. In this respect, only the core of the program generator is implemented by hand, and afterwards, this core provides automation for additional features of the VIATRA model transformation system.

Despite the theoretical well-foundedness of Prolog, in many practical cases, UML CASE vendors would require a tighter integration of transformers than provided by such Prolog programs executed in a third-party prototype tool (such as VIATRA). Therefore, an additional model generation approach [185] (Sec. 5.4) targets to yield object-oriented transformation programs in the form of standard *Action Semantics (AS)* [109,123] expressions. This language was originally created as a standard means to specify methods and actions within UML; however, it also provides a general and implementation (or programming language) independent virtual machine for manipulating UML models. As a result, automated model transformations can be carried out within UML CASE tools (like BridgePoint [139], or iUML [95]) having support for the AS standard.

Note, in the meanwhile, that even though Action Semantics is a standard representation, due to the lack of concrete visual syntax (i.e., the concrete syntax of AS expressions are tool-dependent, only their abstract syntax is standardized) it is less intuitive as a high-level specification language than visual graph transformation rules, which fact serves as an additional argument for our automatic generation approach.

### 1.4.6 The transformation engine

Therefore, the transformation engine that execute the previously generated transformation scripts is totally different in case of transformers written in Prolog or AS.

In the Prolog case, the XMI based models and rule descriptions are translated into a Prolog notation serving as the input data and the program to be executed, respectively. In this sense, we have a *standard (XMI) representation of models* and transformation rules, but the automatically generated transformation programs are tool-specific.
On the other hand, in the AS case, the generated transformers are standardized and they are executed on the virtual machine provided by different UML tool vendors to manipulate the internal, tool-dependent representations of UML models (following the UML standard up to a certain extent).

Further model transformation approaches frequently use XSLT (eXtensible Stylesheet Language Transformation) [159], which is an XML technology used for describing transformations between XML files. Note, however, that XSLT is designed for mainly syntactic manipulations on simple XML trees, and it does not scale up well (neither in design time, nor in run-time performance) for industrial size mathematical model transformations with complex graph manipulations as shown by several experiments carried out in student projects (see also Sec. 5.1.1).

### 1.4.7 Benchmark model transformations

The feasibility of our approach was demonstrated on various industrial size model transformations implemented and executed within the VIATRA framework (see Chapter 8 for several benchmarks).

The pilot UML transformations (i.e., transformations having UML as the source language) include (i) formal proof of logic correctness of UML statechart specifications [105], (ii) check of their completeness and consistency [132], (iii) quantitative dependability evaluations [25, 68], and (iv) high-level fault modeling and analysis. These performance of model transformations was tested on industrial models including the UML model of an artificial kidney system, a railway supervisory traffic control and optimization system, and a radio transmission protocol.

According to our experiments, the time required for the transformation is just a few percentage of the total time spent on the formal analysis exploring the entire state space as model transformations executed on these industrial models generated the input language of the target analysis tool within a minute.

Additional pilot model transformations included our reflective model transformation based automated program generation approach [167] itself (which, in turn, consists of several model transformation steps), and a general open, SVG-based visualization framework for modeling languages defined by metamodels [57].

Many of these transformations were conceptually designed by students as part of their own PhD thesis, we carefully selected benchmark transformations for demonstration purposes (in Chap. 8) that do not collide with their upcoming thesis.

### 1.4.8 Back-annotation of analysis results

The results of the mathematical model transformation are planned to be automatically back-annotated to the UML based system model. Thus, the system analysts are reported from conceptual bugs in their well-known UML notation. Unfortunately, the current version of UML does not directly support the representation of analysis traces. For instance, the sequence of fired statechart transitions that leads to a deadlock according to the verification tool lacks a fine-grained UML representation (using UML sequence diagrams for that purpose is a hack).

Moreover, as model transformations are frequently projections in a mathematical sense, they cannot be inverted in general. Furthermore, several formal analysis methods often perform another model transformation (e.g., a deadlock detection algorithm may take the description of a transition system as input and may generate a sequence of fired transitions as output). For this reason, back-annotation is not equivalent with an inverse model transformation, as it only requires the identification of related source and target objects.

Certain technological problems arise due to the lack of proper handling of XMI by UML CASE tools. In theory (i.e., in their advertisement), these UML tools provide facilities for exporting and importing UML models in XMI format. However, up to now, model interchange between different tools simply has not worked in practice (i.e., model import typically failed) One of the main reasons
for that is that only the abstract syntax of UML models has a standardized XMI representation but UML still lacks such a format for diagram interchange.

Therefore, we only propose a partial solution for the back-annotation problem in the current thesis, while further research (and standardization) is required to achieve full automation in the reverse direction of transformations.

1.5 The Structure of the Thesis

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

- In order to provide a visual but mathematically precise means to specify modeling languages of arbitrary domain, multilevel metamodeling framework (VPM) is introduced in Chapter 2 based upon a refinement calculus in analogy with the structure of mathematical definitions.
- In Chapter 3, I define a meta-level and model-level formal representation of VPM models, and elementary operations based on abstract state machines (ASMs) that provide consistent manipulation of VPM models. In this sense, ASMs will serve as a common semantic basis throughout the thesis.
- Chapter 4 introduces (the theoretical concepts of) a model transformation approach (based on graph transformation), and defines formal semantics for different graph transformation approaches based on abstract state machines.
- Chapter 5 presents automated means to generate the implementation of model transformation programs in the form of a Prolog program or standard Action Semantics expressions where the program generation process itself is also defined by model transformations.
- Chapter 6 proposes a model checking based technique to formally verify models of arbitrary modeling languages (defined by metamodeling and graph transformation rules) by existing model checker tools.
- The correctness and completeness of model transformations are investigated in Chapter 7. There I first propose the use of planner algorithms for proving the static correctness and completeness of model transformations. Then an automated model checking based technique is discussed to verify that model transformations preserve (dynamic) semantic properties.
- The feasibility of the our entire model transformation approach is demonstrated in Chapter 8 with several benchmarks on specifying modeling languages (including Petri nets and a simplified fragment of UML Statecharts based on Extended Hierarchical Automata (EHA) and model transformations (including an industrial strength transformation from EHA to Petri nets).
- Finally, Chapter 9 concludes the main parts of the current thesis.

A brief summary of the two appendices is as follows.

- Appendix A provides the proofs of theorems and propositions appearing throughout the thesis.
- Finally, Appendix C complements Chapter 6 by listing the model checker code for the running example of that chapter.

For reading all these chapters, (at least) a basic knowledge of UML is required.

Notation Guide

In order to obtain a consistent appearance of the thesis, the following regulations are followed.

- This thesis is mainly written in third person singular. In conclusions after each chapter, I stress my own contribution by using first person singular or plural.
14 1 Introduction

- New **concepts** are printed in **bold** letters (typically for their first use).
- All kinds of **definitions** and **theorems** (formal, semi-formal and informal) are emphasized hence printed in **italics**.
- Code **extracts** are **typewritten**.
- For referring to **text in figures**, **sans serif** letters are used.
- The most of the new concepts are indexed and collected hierarchically at the very end of the thesis.

In the next chapter, I introduce a general, multilevel metamodelling framework that serves as the underlying theoretical background for capturing the static structure of modeling **languages**.
VPM: Mathematics of Metamodelling is Metamodelling Mathematics

In the current chapter, the main concepts of MOF metamodelling are overviewed first. MOF metamodels provide the standard way to define the abstract (static) syntax of modeling languages. However, I demonstrate that the MOF standard has certain weaknesses concerning its expressiveness and precise semantics.

I propose VPM, a visual but mathematically precise multilevel metamodelling framework with formal semantics based on a refinement calculus where the structure of modeling languages taken either from mathematical or systems engineering models can be defined (i) in analogy with the structure of mathematical definitions, however, (ii) in a UML-like notation without cumbersome mathematical formulae.

In addition, a static consistency analysis technique is introduced to automatically detect (and partially correct) contradictions in the refinement hierarchy during the evolution of either a model or a modeling language.

2.1 Motivation: Metamodelling and Mathematics

Previous research (in project HIDE [24]) demonstrated that automated transformations of UML models into various semantic domains (including Petri nets, Kripke automata, dataflow networks) allow an early evaluation and analysis of the system. However, preliminary versions of such transformations were rather ad hoc resulting in error prone implementations with an unacceptably high cost (both in time and workload). A main reason for that relies in the fact that the source and target modeling languages of transformations were not handled precisely and uniformly. While UML (as the source language) were handled relatively precisely based upon the standard UML metamodel, modeling languages taken from mathematical domains (i.e., target languages of transformations) were described in an ad hoc way.

In theory, the precise definition of a visual modeling language requires the specification of

- the **abstract syntax** of the language capturing the main conceptual elements of the problem domain (such as states or transitions as in case of UML) and their interconnections with mathematical preciseness;
- the **concrete syntax** of the language which defines the visual elements depicting the conceptual elements of the language (e.g., a state is a rounded rectangle while a transition is an arrow);
- the (static) **well-formedness constraints** of a language which define additional restrictions on valid (well-formed) instance models of the language (e.g., a composite state in UML statecharts is allowed to have a single initial pseudo state)
- the **dynamic semantics** of the language, which describes (i) the meaning of conceptual elements (called denotational semantics), (ii) the evolution (or behavior) of model instances (called operational semantics), or (iii) constraints on operations (called axiomatic semantics).
The abstract syntax and well-formedness constraints together are frequently called as the static structure of the language. Unfortunately, there is a still a certain dichotomy between metamodeling and mathematics especially in industrial settings where the term “metamodeling” frequently refers only to the definition of the static structure. For that reason, we postpone the specification the dynamic semantics of a modeling language, and in the current chapter we focus merely on how to specify the static structure of a modeling language by metamodeling techniques with mathematical preciseness.

Below we briefly summarize first how metamodeling and mathematics influenced each other.

Mathematics of metamodeling

When regarding the precise semantics of UML (or metamodeling), one may easily find that there is a huge contradiction between engineering and mathematical preciseness. UML should be simultaneously precise (i) from an engineering point of view to such an extent adequate to engineers who need to implement UML tools but usually lack the proper skills to handle formal mathematics (as provided by the MOF standard), and, (ii) from a mathematical point of view necessitated by verification tools to reason about the system rigorously.

The UML 2.0 RFP requires (votes for) engineering preciseness: “UML should be defined without complicated mathematical formulae.” However, when considering model transformations of UML sublanguages into executable platform/code or appropriate semantic domains (i.e., the abstract syntax of a UML model is mapped into such as Petri nets, finite automaton, etc.), the proper handling of formal mathematics is indispensable for developing automated and highly portable tools for analysis and code generation in the MDA environment.

Metamodeling mathematics

Meanwhile, recent standardization initiatives (such as PNML [3], GXL [150], GTXL [157], or MathML [197]) aim at developing XML based description formats for exchanging models of mathematical domains between different tools. Frequently (as e.g. in [157]), such a document design is driven by a corresponding UML-based metamodel of the mathematical domain. However, improper metamodeling of mathematics often results in conceptual flaws in the structure of the XML document (e.g., in PNML, arcs may lead between two places, which is forbidden in the definition of Petri nets).

On the other hand, as demonstrated in [50] (where dependability analysis of BPM-based e-business applications is carried out with dataflow networks as the mathematical background), a well-constructed metamodel can drastically reduce the time and workload related to the implementation of even complex analysis tools.

Problem statement

As a first prerequisite for a complex model transformation system, modeling languages taken from either engineering and mathematical domains should be handled uniformly and mathematically precisely.

Unfortunately, the current Meta Object Facility (MOF) standard fails to achieve this goal since the following conceptual problems (discussed in details later in Sec. 2.3) can be identified.

- **Lack of metamodel reuse.** MOF fails to provide a means for the reuse of metamodels by appropriate inheritance mechanisms. As a result, core (abstract) parts are reintroduced over and over again in different standards defined by MOF metamodels. Since inheritance is only defined for (meta)classes, the reuse of abstract modeling concepts (like queues, trees, etc.) can only be specified by additional constraints but not on a metamodel basis.

- **Structural redundancies.** Even the UML 2.0 RFP identifies that a core metamodeling language should be very concise. Both current MOF and the upcoming UML 2.0 metamodeling kernel contain concepts that are redundant from a mathematical point of view.
• **Lack of multiple instantiation.** Meanwhile multiple inheritance is quite common in MOF, multiple instantiation is disallowed which decreases its expressiveness.

• **Problems with metalevels.** As demonstrated in [10] there are fundamental problems with the handling of metalevels in the four-layer metamodeling architecture of MOF. The main problem here is that MOF metamodeling is constrained to exactly four metalevels (type-instance relations), however, a multilevel solution would indispensable in a general metamodeling framework (supporting an arbitrary number of type-instance relations).

**Related work**

Unfortunately, none of the existing semi-formal and formal metamodeling approaches are able to overcome the previous problems. Below we summarize the deficiencies of leading metamodeling approaches.

• The underlying metamodeling basis of generic off-the-shelf metamodeling tools (like DOME [88] or MetaEdit+ [110]) offer a certain level of metamodel reuse but the rest of the problems are left unhandled.

• BOOM [126] is a framework for formal specification of object-oriented modeling languages using the textual description language ODAL. Unfortunately, BOOM only contributes to the reuse of metamodels.

• BON (Business Object Notation) [130] has its own notation for both constraints and metamodels, and the concepts are formalized in PVS [48]. Only reuse of metamodels are supported to a certain extent this time as well.

• GME (Generic Modeling Environment) [107] introduces the concepts of implementation and interface inheritance to allow the composition of metamodels. This is a novel contribution only to the problem of metamodel reuse.

• PROGRES [151] is a general graph transformation tool built upon a precise metamodeling framework based on graph schemata [149]. Here an instance graph is mapped to its “metamodel” graph schema by typing graph homomorphisms, which supports an arbitrary number of metalevels. PROGRES supports association (edge) inheritance but does not support the inheritance of entire metamodels (see MMF below).

• MMF (MetaModelling Framework) [38] introduces the concept of package inheritance for metamodel reuse, and has the potential to support multilevel metamodeling [7]. However, MMF is still redundant, moreover, multiple instantiation is also excluded. Moreover, as shown in [10], the MML approach also has the problem of “shallow instantiation” and “the replication of concepts”.

• Finally MoMM (Metamodel for Multiple Metalevels), the proposal of [10] overcomes these problems and defines a very concise multilevel metamodeling core by assigning “potencies” and “levels” to model elements that provide deep instantiation of classes. The problem with that solution is that it does not handle multiple instantiation, moreover, the possibilities of metamodel reuse are not addressed.

**Own contribution**

After a brief informal introduction to the concepts of metamodeling (in Sec. 2.2), I demonstrate in details (in Sec. 2.3) that meanwhile the overall goals of MOF metamodeling are highly relevant for the specification and integration of modeling languages, the traditional metamodeling foundations and concepts of MOF (like the four-layer architecture itself) are inappropriate from many aspects.

As the main contribution of the current chapter (Sec. 2.4), I propose a visual but mathematically precise metamodeling framework (abbreviated in the sequel as VPM: Visual and Precise Metamodeling) based upon the structure of mathematical definitions for defining the abstract syntax of modeling

---

1 At least in theory, as it is actually limited to the traditional four levels in the PROGRES environment.
languages. Starting from a very concise (thus easy-to-implement) kernel language VPM builds up a hierarchy of models and modeling languages satisfying the rules of a refinement calculus that handles the most important features of the current (and upcoming) metamodeling standard but avoids the problems identified in Sec. 2.3.

In addition, I present a static consistency analysis technique (in Sec. 2.5) to automatically detect (and partially correct) contradictions in the refinement hierarchy during the evolution of either a model or a modeling language.

For a better comprehension, the major information flow summarizing the organization of sections in the current chapter is depicted in Fig. 2.1. In order to understand upcoming parts of the thesis a special emphasis should be put on reading the highlighted sections (i.e. Sec. 2.2 and 2.4.2)

2.2 Specifying the Abstract Syntax of Modeling Languages

Initially, we summarize the major concepts for defining models and modeling languages by the traditional MOF metamodeling techniques. As the MOF metamodeling terminology is being merged with the core constructs in the upcoming UML 2.0 standard, the terms of (i) UML class diagrams and MOF metamodels, and (ii) UML object diagrams and MOF instance models will be used interchangeably, even though they can be very different from an application point of view (i.e., metamodels and models are related to language design while class diagrams and object diagrams appear in application development).

Note that for the current section, we relax the precise mathematical treatment of metamodeling until the end of this section and an informal conceptual overview is given first to increase the legibility and understandability for non-experts.

2.2.1 Metamodels

The abstract syntax of domain specific modeling languages is defined by a corresponding metamodel, which conforms to the best engineering practices in visual specification techniques. The main concepts of metamodeling are demonstrated on a small example by defining the modeling language of finite automata in Fig. 2.2.

Example 2.1 (Metamodel of finite automata). According to the metamodel, the language of finite automata consists of the class Automaton, which is in turn composed of (as denoted by the aggregations with the arrow with a black diamond) classes State and Transition. A transition is leading between its single from state and to state as defined by the corresponding references (depicted as arrows) and


**multiplicity** constraints (number 1 appearing next to the arrow). The initial states of the automaton are marked with $init$, and there may be several initial states in an automaton (as defined by the multiplicity $\ast$).

The active states are marked with $current$ association, while the reachable states starting from the initial states are modeled by $reachable$ associations. They are denoted differently (i.e., as dashed red arrows) to emphasize that they contribute to the dynamic behavior of finite automata.

The construct $FAElement$ (a class with a name printed in italics) is an abstract class, which means that no instantiations are allowed from it. However, by generalization (traditional object-oriented “is-kind-of” relations denoted as arrows with white arrowhead), the name attribute related to $FAElement$ may be inherited (and thus implicitly reused) in subclasses (like Automaton, State or Transition).

**Definition 2.2 (Metamodel).** As a summary, a metamodel $MM$ consists of the following elements.

- **Classes** are used for identifying entities of the modeling language. Abstract classes are classes that must not have instances. Subclasses of a class can be derived by generalization; in this case an instance object of a subclass is an instance of the superclass as well, moreover, the instance of the subclass inherits the structure of (instance of) the superclass.

- **Attributes** are value holders in the instances of a class.

- **Associations** are binary relations between class instances. An association is either an aggregation (when a class instance is composed of several other class instances) or a simple reference.

In addition, both ends of an association may have a multiplicity constraint attached to them, which declares the number of objects that, at run-time, may participate in the association. Common multiplicity values are the following.

- $1$: Exactly one object will participate;
- $\mathit{n}$: Exactly $n$ objects will participate;
- $0..1$: At most one (i.e., zero or one) object will participate;
- $m..n$: The number of participating objects are between $m$ and $n$;
- $\ast$: Zero or more objects will participate.

However, while all the previous concepts define extensions to the modeling language (i.e., new elements), multiplicities are restrictions, thus formally they are part of the static constraint language. Therefore, we will frequently omit those multiplicity constraints from metamodels to emphasize the mathematically sound modeling practice (not followed by the MOF standard, unfortunately), i.e., a metamodel is purely declarative, and anything restrictive is part of the constraint language.

**Static and dynamic model elements**

MOF metamodels only contain semi-formal specification of the static structure (i.e., abstract syntax and well-formedness constraints) of a modeling language, while the dynamic behavior of model
instances are only informally defined. For instance, the UML specification explicitly introduces state-chart transitions as a (meta-)class, but the notions of enabled or fireable transitions are not parts of the metamodel even though the dynamic semantics of UML statecharts would highly rely on them. This fact largely hinders the development of UML CASE tools that implement a statechart virtual machine as the tool developers only have a standard guideline concerning the static structure of statecharts but not concerning the dynamic behavior.

However, as we aim to provide a complete formal description of visual modeling languages modeling language concepts required for specifying the dynamic semantics should be handled as well.

**Proposal 2.3 (Metamodeling of dynamic model elements).** The metamodel of a visual modeling language should simultaneously and explicitly contain but clearly separate static and dynamic model elements [168].

By dynamic model elements we mean elements that can be altered, (updated, removed or added) during the execution of models, which can be easily collected by analyzing the structure of the behavioral specification (graph transformation rules in our case).

For a notational guidance, dynamic elements will appear in red in models and metamodels with additional dashed lines in case of associations.

**Example 2.4 (Dynamic and static concepts in finite automata).** In case of finite automata (as it will turn out later in Fig. 4.6), associations current and reachable are dynamic since (i) the current states of an automaton may change through the execution of the automaton and (ii) the set of reachable states (marked by reachable associations) are also evolving.

In many cases, the precise definition of a modeling language requires to introduce some auxiliary or derived concepts. For instance, when inducing a state hierarchy for finite automata (like UML statecharts, for instance), the metamodel will typically define only the direct substate (child state) association between states. However, for defining the dynamic semantics of the language (see Sec. 8.2 for the case study on UML statecharts), we will also need the concepts of indirect substates (descendant states), which can be computed mathematically as a transitive closure of the previous one. However, as we introduce such a new notion in the metamodel as well.

**Proposal 2.5 (Metamodeling of derived model elements).** The metamodel of a visual modeling language should explicitly contain derived (auxiliary) model elements. By derived (auxiliary) model elements we mean elements that might not be present in a compacted static representation of the model but they can be derived (e.g., by an additional set of graph transformation rules in our case) prior to the execution of a model; however, they never change during execution.

Such a clear separation of derived static concepts and their dynamic interpretation (proposed first in [168]) scales up well for rapidly changing modeling standards like UML since it better highlights potential future changes in the standard than purely textual remarks (moreover, it allows a comparison, for instance, of different statechart variants).

### 2.2.2 (Instance) Models

We continue our finite automaton example with a well-formed (instance) model of the language (i.e., one that corresponds to the metamodel).

**Example 2.6 (A sample finite automata).** A sample automaton object $s_1$ consisting of three state objects ($s_1$, $s_2$, $s_3$) and three transitions between them $t_1$ (leading between $s_1$ and $s_2$), $t_2$ (leading between $s_2$ and $s_3$), and $t_3$ (leading between $s_1$ and $s_3$) is depicted in Fig. 2.3.

The object $s_1$ contains a name slot with the concrete string value "$d$".
We can notice from the \textit{init} link that the initial state of \texttt{s1} is \texttt{s1}. As the life-cycle (execution) of the automaton has not started yet, no states are marked with \textit{current} or \textit{reachable} links.

We use \texttt{s1:State} to denote that the type of object \texttt{s1} in the instance model corresponds to class \texttt{State} in the metamodel, and we say that \texttt{s1} is of class (type) \texttt{State}. In case of irrelevant identifiers, only the classes of objects (associations of links) are printed after the colon (:).

Instances models can be represented in their abstract syntax (see the automaton in the left of Fig. 2.3) or in their concrete syntax (in the right of Fig. 2.3). In general, a model depicted in the concrete syntax is more legible for humans, while a model represented in its abstract syntax is “more legible” for tools. Despite the fact that more humans are expected to read the current thesis than tools (naturally, not counting the \texttt{EjXprocessor}), examples will typically be provided in the abstract syntax, which better emphasizes the class-instance relationship between models and metamodels.

\textbf{Definition 2.7 (Instance model).} As a summary, an instance model \( M \) consists of the following elements.

- \textbf{Objects} are uniquely identified instances of (non-abstract) metamodel classes.
- \textbf{Links} are instances of metamodel associations (uniform for both references and aggregations) that interconnect existing objects.
- \textbf{Slots} are value holders for metamodel attributes in existing objects.

\textbf{Definition 2.8 (Type conformance).} A model \( M \) is an instance of metamodel \( MM \) if

- For all object \( obj \) in \( M \), its direct type exists as a class \( cls \) in \( MM \).
- For all slots \( sl \) of an existing object \( obj \), there exists a class \( cls \) in \( MM \), which has an attribute \( att \) corresponding to the type of \( sl \), and the class is the direct type (or a supertype) of \( obj \).
- For all links \( link \) leading from object \( obj_{from} \) to object \( obj_{to} \), there exists an association \( assoc \) in the metamodel \( MM \) which leads from class \( cls_{from} \) to class \( cls_{to} \), and \( cls_{from} \) is a supertype of \( obj_{from} \), and \( cls_{to} \) is supertype of \( obj_{to} \).

For most parts of the current thesis, we will restrict our models in addition in such a way that \textit{only one link of a certain type may lead between two objects}, which implies that links are relations between objects.

\textbf{Example 2.9.} Figure 2.4 demonstrates the correct interpretation of type conformance of links in a graphical way.

Here the \textit{link} leading between objects \texttt{a1} and \texttt{b1} is an instance of association \texttt{assoc} since (i) the class \texttt{A} of object \texttt{a} is a subclass of class \texttt{SuperA}; (ii) the class \texttt{B} of object \texttt{b} is a subclass of class \texttt{SuperB}; and (iii) association \texttt{assoc} is leading between classes \texttt{SuperA} and \texttt{SuperB}. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2_3.png}
\caption{A sample model of finite automata}
\end{figure}
Formalization of metamodeling: typed graphs

Metamodels and instance models are frequently (although not in the current thesis) formalized as typed and attributed graphs. The concept of typed graphs [44] captures the well-known dichotomy between classes and objects, or between database schema and instance, in the case of graphs. All classes are mapped into a corresponding node type, and all associations are projected into an edge type. The inheritance hierarchy of metamodels can be preserved by an appropriate subtyping relation on nodes. Class attributes are derived into graph attributes where the latter may be treated mathematically as (possibly partial) functions from nodes to their domains.

Note in the MOF (and UML standard), uni-directional associations (like the ones depicted in Fig. 2.2) restrict the navigability of association instances. However, for the current thesis, such arrows only demonstrate that models and metamodels can easily be represented as graphs and we implicitly suppose that (i) either both ends a graph edge (modeling an association) are implicitly navigable or (ii) we can explicitly split bi-directional associations into two and introduce two directed edges. In other terms, navigability issues are irrelevant for our investigations.

Concerning the traditional formalization by typed graphs, objects and links between them are mapped into nodes and edges, respectively, in the (model) instance graph. Each node and edge in the model graph is related to a corresponding graph object in the type graph by a corresponding typing homomorphism. Slots associated to objects can be interpreted as attributes on graph nodes.

2.2.3 The four layer MOF metamodeling architecture

The concepts of four layer MOF metamodeling architecture originate in the need for an effective design process of formal specification and modeling languages. The large number of similar languages - often supported nowadays by visual diagrams - necessitates a common model description language (called meta-metamodel or MOF Model).

The sentences of this top-level language (denoted as metamodels) are to describe the structure of domain specific information models. For instance, the metamodel of UML is to describe the major concepts of UML. However, these sentences of the common meta-metamodel may be regarded in turn as separate sub-languages (the language of UML, EDOC, BPM, etc.), thus they provide a grammar to describe sentences of a lower meta-level. These lower level sentences are denoted as models (e.g., the UML metamodel serves as a common grammar for describing different UML models as sentences).

As a result, a model hierarchy (summarized in Table 2.1) is available with four meta-layers.

<table>
<thead>
<tr>
<th>Meta-level</th>
<th>MOF terms</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>meta-metamodel</td>
<td>The MOF Model</td>
</tr>
<tr>
<td>M2</td>
<td>metamodel</td>
<td>UML Metamodel</td>
</tr>
<tr>
<td>M1</td>
<td>model</td>
<td>UML Models</td>
</tr>
<tr>
<td>M0</td>
<td>data</td>
<td>modeled systems</td>
</tr>
</tbody>
</table>

Table 2.1. The four-layer MOF metamodel architecture
Although, the metamodeling concepts are traditionally related mainly to UML and software modeling languages, the similar concepts can be applied for describing the structure of arbitrary mathematical models as such models use languages of lower abstraction level. A metamodel of e.g., Petri Nets, or finite automata provides an easy-to-understand way to obtain a visual overview of the underlying mathematical structures.

2.3 Problems of MOF Metamodeling

At first, we briefly identify (or revisit) some major problems of MOF metamodeling that hinder the use of MOF as the ultimate technique for specifying modeling languages of arbitrary domains in a hierarchical and reusable way.

2.3.1 Lack of package (metamodel) inheritance

Many existing UML profiles clearly demonstrate that the most general concepts (like events, actions, constraints, basic types, etc.) are redefined over and over again for many different profiles. This problem relies in the fact that MOF metamodels cannot be arranged in a refinement hierarchy, thus domain experts responsible for the creation of a specific profile cannot build upon a reusable abstract metamodel library. Such a metamodel hierarchy is in analogy with meta-level design patterns that would encapsulate and reuse best engineering (and mathematical) practice in language design. In fact, a proper metamodeling technique would simultaneously handle both meta-level and model-level design patterns.

The key notion of such a hierarchy is captured as package inheritance in the MML approach [38], which extends the inheritance mechanism of classes to entire metamodels (encapsulated as a package). However, we demonstrate in Sec. 2.4.1 that the underlying concepts come from far beyond, namely, from the structure of mathematical definitions where a new notion is defined on the basis of an existing one (with the Zermelo-Frankel set theory on the top to provide meta-circularity in the engineering sense). For instance, each mathematical textbook on graph theory first introduces the notions of a graph, and then different subdomains (like bipartite graphs, planar graphs, etc.) are derived by restrictions (e.g., “a bipartite graph is a graph with...” as expressed in Fig. 2.5).

![Fig. 2.5. Metamodel (package) inheritance: a bipartite graph is a graph](image)

Fortunately, the latest proposal for the new UML 2.0 standard [162] seems to provide some advanced concepts (such as redefine, import, package merge) to capture such a reusable metamodel hierarchy.

2.3.2 Lack of association inheritance

As MOF (and UML) evolved from traditional object-oriented programming languages only the inheritance of classes is allowed. However, the lack of an inheritance concept for associations hinders
the development of reusable metamodel template libraries, since the domain expert has to write additional well-formedness constraints (e.g., in OCL) to express that associations in the library should be restricted to lead between the derived classes of the new metamodel. Unfortunately, this is very error prone when compared to a proper inheritance mechanism for associations, since if such well-formedness constraints are omitted, one might not be able to detect that an instance model does not conform to its metamodel.

The metamodel of queues in Fig. 2.6 specifies that a Queue object may contain elements of class QElem. Now, if queues are aimed to be reused to contain only integers, then one can state in MOF that an integer queue IntQueue is queue, and the class Integer is a subclass of QElem, but without association inheritance, the fact that an integer queue may only contain integers as elements can only be expressed by explicit OCL constraints.

2.3.3 Structural redundancies in MOF

Even though MOF should provide a minimal set of kernel constructs that are required to specify modeling languages in a hierarchical way, both the current MOF and the UML 2.0 core is redundant concerning containment and attributes.

As demonstrated in Fig. 2.7, one can express the mathematical fact that name is a function that maps States to Strings as (i) introducing name as an attribute of class State, or (ii) using an association with exactly one (or at most one in case of partial functions) multiplicity at the navigable String end.

Moreover, expressing containment for classes is also redundant as we can alternatively use package containment or aggregations (for specifying that, for instance, a Statemachine contains States). In many cases (concerning reusable metamodels), it is extremely hard to judge whether a package or a class (or probably both) is required for a certain concept.

2.3.4 Lack of multiple instantiation

MOF (and UML) follows traditional type theoretic foundations of object-oriented programming languages where even if multiple inheritance is allowed, objects are only permitted to have a single class as their direct type.

However, the right part of Fig. 2.8 depicts a simple example to highlight the essence of the problem. Let us suppose that accepting states AccState and initial states InitState are two subclasses of State in the language of finite automata. Then a state instance which is simultaneously accepting and initial
can only be created if an object is allowed to have multiple types. In Sec. 2.4.2, we demonstrate that multiple instantiation can be handled identically to multiple inheritance.

As an additional example, on the left of Fig. 2.8, a football team with is captured where goalkeepers and strikers are modeled by separate classes. However, if a player may have totally different roles on the pitch (see Jorge Campos, the former multi-functional Mexican goalkeeper for a real example), we must create two instances of the same real-world entity, which is somewhat contradictory.

Unfortunately, the same problem appears in large in the UML environment as the entire extension mechanism of UML (based on profiles and stereotypes) is chaotic (see [11] for an overview of the diversity of proposals that formally capture stereotyping) due to the lack of multiple instantiation. In principle, a UML profile (like SPEM [117], EDOC [118], or GRM from the UML Profile for Schedulability and Time [119]) is a modeling language (metamodel) designed for certain domain created by experts in order to model the target application from additional aspects. However, such a metamodel itself is totally independent of UML, in other terms, UML is just one language that a domain profile could be tailored to but could possible be reused in other modeling languages. For instance, resource usage can be modeled by the GRM formalism [119] not only for a software application (embedding it in UML) but also for control applications (e.g. combining with Matlab/Simulink).

A proper metamodel-based handling of the problem is to instantiate constructs in a user model from multiple modeling languages.

2.3.5 Problems with metalevels

As discussed previously in [10], there are fundamental problems with the traditional four-layer MOF architecture. In many cases, same concepts are replicated both on meta-level and model-level (or other two adjacent layers) as (meta)classes can only be instantiated one level down (up?) in the hierarchy (shallow instantiation).

In the paper (see also Sec. 2.4.2), we argue that the problem is directly caused by the fact that the metamodel derivation process is quantized into four discrete levels, moreover, the borders of such metalevels are fixed. As a result, one has to artificially distinguish between classes and objects (metalevel and model-level instances) of the same real-world entity, which doubles the size of the model space.

The problem relies in the fact that during different modeling phases, the same concepts can be regarded both classes and instances as well. For instance, when modeling databases in Fig. 2.9, we can simultaneously say (typically, on different level of abstraction) that a relational database RelDB is both a subclass and an instance of databases (see the generalization and instance-of relations leading to Database). According to the traditional MOF concepts, we need to create a separate class and object for the same real-world concept.
2.3.6 Metamodeling vs. software engineering

As a conclusion, many of these problems are probably not crucial in a general purpose modeling language (like UML) for designing software applications (as they might be too abstract for an average systems engineer). However, they demonstrate major weaknesses of a metamodeling kernel language used for designing other modeling languages by domain experts. Unfortunately, many of the previous problems are left unhandled even in the new UML 2.0 core language [162].

As UML has been used in practice for many years now with a wide range of existing applications, resolving such problems with minimal changes in the standard is very difficult. In fact, there are two rather complimentary conceptual solutions: (i) one is to separate the techniques for designing applications and modeling languages, which may keep UML relatively unaltered but contradicts with the MDA vision (saying that models and modeling languages are designed within the same modeling approach, i.e., UML), (ii) the other is to adapt the changes in UML as well, which would result in a major redefinition of (at least) its infrastructure.

Although we propose rather radical changes to the underlying metamodeling concepts, an intuitive and simplified UML/MOF notation is used in the paper to emphasize that changes in the depth (in the semantics of metamodeling) do not necessarily involve changes on the surface (in the syntax of metamodels), and if so, these changes are mainly simplifications of the existing MOF standard. Moreover, as VPM is a multilevel approach the original MOF metamodel can be integrated into our framework as any other modeling languages thus the conformance with the current version of the standard can also be maintained.

2.4 VPM: Structural Refinement of Metamodels

Below we define a structural refinement calculus on set theoretical basis (i.e., refinement of sets, relations, functions and tuples) for major MOF (UML) constructs. Our metamodeling framework is gradually extensible in depth, thus it only contains a very limited number of core elements, which highly decreases the efforts related to implementation. Moreover, in order to avoid the previous metamodeling problems we introduce dynamic (or fluid) metalevels where the type-instance relationship is derived between models instead of explicitly predefining it by (meta)levels. Our approach has the major advantage that the type-instance relations can be reconfigured dynamically throughout the evolution of models, thus transformations on (traditional) model and metamodel “levels” can be handled uniformly.

2.4.1 Visual definition of Petri nets

Before a precise and formal treatment, our goals are summarized informally on a metamodeling example deliberately taken from a well-known mathematical domain, i.e., Petri nets. Petri nets are widely used means to formally capture the dynamic semantics of concurrent systems. However, due to their
easy-to-understand visual notation and the wide range of available tools, Petri net tools are also used for simulation purposes in industrial projects (reported recently, e.g., in [152]).

**Definition 2.10.** A simple Petri net $PN$ is a bipartite graph with distinct node sets $P$ (places) and $T$ (transitions), edge sets $IA$ (input arcs) and $OA$ (output arcs), where input arcs are leading from places to transitions, and output arcs are leading from transitions to places. Additionally, each place contains an arbitrary (non-negative) number of tokens.

Now, if we assign a UML class to each set of this definition (thus introducing the entity of Place, Transition, InArc, OutArc, and Token), and an association for each allowed connections between nodes and edges (connections such as fromPlace, toPlace, fromTrans, toTrans, and tokens), we can easily obtain a metamodel of Petri Nets (see the Petri Net package in the upper right corner of Fig. 2.10) that seems to be satisfactory.

![Graph](image)

**Fig. 2.10.** Defining the structure of Petri Nets

However, we have not yet considered a crucial part of the previous definition, which states that a Petri net is, in fact, a bipartite graph. For this reason, after looking up a textbook on graph theory, we may construct with the previous analogy the metamodel of bipartite graphs (depicted in the lower left corner of Fig. 2.10) with ‘boy’ and ‘girl’ nodes\(^2\), and ‘boy-to-girl’ (BG) and ‘girl-to-boy’ (GB) edges. Moreover, if we focus on the fact that every bipartite graph is a graph, we may independently obtain a metamodel of graphs (see the upper left corner of Fig. 2.10).

First, we intend to inter-relate these metamodels in such a way to be able to express that, for instance, (i) the class Node is a supertype of class Boy, and (ii) the association fromPlace is inherited (indirectly) from the association from. As a result of such elementary inheritance relations, we would also like to state that the metamodel of bipartite graphs is a generalization of the metamodel of Petri nets. In the rest of the paper, we denote these relations uniformly by the term refinement, which simultaneously refers to the refinement of entities, connections, and (meta)models.

Our notion of refinement should also handle the instantiations of classes. For instance, in the SimpleNet package in the lower right corner of Fig. 2.10, a Petri net model consisting of a single place

\(^2\) Bipartite graphs are often explained as relations between the set of boys and girls.
with one token is depicted. This model is regarded as an instance of the Petri net metamodel as indicated by the dashed arrow between the models.

From a practical point of view, supposing that we have an extensible metamodel library, a new metamodel can be derived from existing ones by refinement. Our main goal is to show that (i) mathematical and metamodel constructs can be handled uniformly and precisely (see Sec. 2.4.2), and (ii) the dynamic operational semantics of models can also be inherited and reused with an appropriate model refinement calculus (see later Sec. 4.5) in addition to the static parts of the models.

2.4.2 Formal semantics of static model refinement

Modeling concepts

Our VPM metamodeling framework uses a minimal subset of MOF constructs (i.e., only classes, associations, attributes, and packages) with precisely defined semantics, which has a direct analogy with the basic notions of mathematics, i.e., sets, relations, functions, and tuples (where tuples are constituted in turn from sets, relations and other tuples).

However, in order to avoid clashes between notions of MOF and set theory as much as possible, a different naming convention is used in the thesis, which simultaneously refers to UML and mathematical elements. A model element in VPM may be either an entity, a connection, or a mapping (see the MOF metamodel of our approach in Fig. 2.11). A unique identifier (accessed by a .id postfix in the sequel) and a set including the identifier of the model element and the identifiers of all the (intended) refinements of the element (accessed by a .set postfix) are related to each of this constructs. For mathematical reasons, the set associated to a model element should also contain the identifier of the element (to be able to detect circularities in typing later on).

An entity E is a set (called as basic entity in this case) or a tuple (denoted as compound entity or model) consisting of sets, relations, functions and tuples (a collection of entities, connections, and mappings, respectively). Entities will be represented visually either by UML classes or UML packages while the notion of containment will be captured by graphical containment (e.g., classes

---

3 This philosophy is in analogy with the axiomatic foundations of set theory. There we have classes as a notion that remains undefined. An element of a class is by definition a set, while the singleton class that contains this element is also a set.
inside a package) or aggregations (leading from entities to both entities, connections and mappings) depending on the context to provide the better match with the conventional notation.

- A **connection** \( R \) between two entities is a binary relation between the associated sets or tuples. Connections are depicted as (directed) associations.

- A **mapping** \( F \) from entity \( E_1 \) to entity \( E_2 \) is a partial function with the domain of (the set of) \( E_1 \) and range of \( E_2 \). Mappings can be denoted visually by an attribute assigned to the entity of its domain with an attribute type corresponding to its range.

A significant change in contrast to [181] is the merging of previously distinct notions of entities and models into a single entity construct, which stems from the fact that a one-dimensional tuple (consisting of only a single set) can be regarded as a set, thus a certain redundancy is eliminated from the underlying mathematical framework of our approach. In object-oriented terms, a uniform class concept is used for both classes and packages (models).

**A refinement calculus for inheritance and instantiation**

The static semantics of our metamodeling framework is based upon a refinement calculus, which uniformly captures the notion of inheritance and type-instance relationship (depicted by UML generalization and instance-of relations, respectively) between arbitrary metalevels *without actually defining the notion of metalevels*.

**Definition 2.11 (Comparison of elements).** A model element \( P \) (i.e., either entity, connection or mapping) is less than (or equal to) a model element \( Q \) (denoted as \( P \leq Q \)) if \( P.\text{set} \subseteq Q.\text{set} \land P.\text{id} \in Q.\text{set} \) thus, if the related set of \( P \) is a subset of the corresponding set of \( Q \) and the identifier of \( P \) is contained by the set of \( Q \), which is an ordering relation combining the subset and set containment relations.

For the notational convention of Def. 2.12, let (i) \( E^{(n)} \) denote an entity consisting of exactly \( n \) subcomponents (in case of \( n = 1 \), the entity is regarded to be basic, otherwise compound) where \( E[i] \) accesses the \( i \)th component (argument) of entity \( E \). \( R(A, B) \) refers to a connection \( R \) between entities \( A \) and \( B \), while (iii) \( F(A, B) \) denotes a mapping with the domain of entity \( A \) and range of entity \( B \).

Our general refinement relation between model elements \( \text{Sub} \) and \( \text{Super} \) will be denoted as \( \text{Sub} \sqsubseteq \text{Super} \) symbol (consistently with the partial order imposed by the refinement graph later in Sec. 2.5), which means that \( \text{Sub} \) is a refinement of \( \text{Super} \) (for the use of our terminology, see Table 2.2). Refinement is unified relation that is either an inheritance \( \rightarrow \) or an instantiation \( \mapsto \) (or both).

<table>
<thead>
<tr>
<th>( \sqsubseteq ): refinement =</th>
</tr>
</thead>
<tbody>
<tr>
<td>inheritance + instantiation</td>
</tr>
<tr>
<td>sub is ... super</td>
</tr>
<tr>
<td>refinement of</td>
</tr>
</tbody>
</table>

\[ \rightarrow\text{: inheritance (subtype)} \]

\[ \mapsto\text{: instantiation ((type-instance relation))} \]

\[ \text{inherited from} \]

\[ \text{generalization of} \]

\[ \text{supertype of} \]

\[ \text{type of} \]

**Table 2.2.** Notation guide for refinement, inheritance and instantiation

**Definition 2.12 (Refinement calculus).** The refinement (\( \sqsubseteq \)) rules of our metamodeling framework (that simultaneously handle inheritance \( \rightarrow \) and instantiation \( \mapsto \)) are as follows.

1. **Basic entity refinement:** \( E_{\text{sub}}^{(n)} \sqsubseteq E_{\text{super}}^{(1)} \) if \( E_{\text{sub}}^{(n)} \leq E_{\text{super}}^{(1)} \), thus if \( E_{\text{super}} \) is a simple entity (one-dimensional tuple only consisting of a set) then refinement is defined as the “less-than” relation.
Informally, $E_{\text{super}}$ is a generalization (or a type of) $E_{\text{sub}}$ (which is either a class or a package) if using a MOF metamodeling analogy.

2. **Connection refinement**: $R_{\text{sub}}(A_{\text{sub}}, B_{\text{sub}}) \subseteq R_{\text{super}}(A_{\text{super}}, B_{\text{super}})$ defined if $R_{\text{sub}} \leq R_{\text{super}} \wedge A_{\text{sub}} \subseteq A_{\text{super}} \wedge B_{\text{sub}} \subseteq B_{\text{super}}$ (where all $A_i$ and $B_i$ are entities). Connection inheritance expresses the fact that MOF associations can also be refined during the evolution of metamodels in addition to the refinement of classes.

3. **Mapping refinement**: $F_{\text{sub}}(A_{\text{sub}}) : B_{\text{sub}} \subseteq F_{\text{super}}(A_{\text{super}}) : B_{\text{super}}$ defined if $F_{\text{sub}} \leq F_{\text{super}} \wedge A_{\text{sub}} \subseteq A_{\text{super}} \wedge B_{\text{sub}} \subseteq B_{\text{super}}$. From a practical point of view, the refinement of MOF attributes is also handled in our metamodeling framework (similarly to classes and associations).

4. **Mapping is connection**: $F(A_{\text{sub}}) : B_{\text{sub}} \subseteq R(A_{\text{super}}, B_{\text{super}})$ if $F \leq R \wedge A_{\text{sub}} \subseteq A_{\text{super}} \wedge B_{\text{sub}} \subseteq B_{\text{super}}$, i.e., functions can be interpreted as special relations. In practical uses of this axiom, traditional cardinality restrictions in MOF metamodels can be strengthened such as the cardinality of a role can be changed from “arbitrary number” ($0..*$) to “at most one” ($0..1$).

5. **Compound (model) refinement**: the refinement of compound entities (or models/packages) is explicitly split into the inheritance and instantiation case, thus $E_{\text{sub}}(n) \subseteq E_{\text{super}}(k)$ defined if $E_{\text{sub}}(n) \rightarrow E_{\text{super}}(k) \vee E_{\text{sub}}(n) \rightarrow E_{\text{super}}(k)$.

a) **Compound entity (model) inheritance** $E_{\text{sub}}(n) \rightarrow E_{\text{super}}(k)$ defined if $E_{\text{sub}} \leq E_{\text{super}} \wedge \forall i \exists j : E_{\text{sub}}[i] \rightarrow E_{\text{super}}[j]$. Informally, there exists a subtype relation for each argument of $E_{\text{super}}$ in a corresponding argument of $E_{\text{sub}}$. In MOF terms, each class in the super package is refined into an appropriate class of the subpackage. However, this latter one may contain additional classes not having origins in the super package.

b) **Compound entity (model) instantiation** $E_{\text{sub}}(n) \rightarrow E_{\text{super}}(k)$ defined if $E_{\text{sub}}[i] \rightarrow E_{\text{super}}[j]$. Informally, there exists a type element for each component of $E_{\text{sub}}$ in a corresponding component of $E_{\text{super}}$. In MOF terms, each object in the instance model has a proper class in the metamodel. However, the metamodel one may contain additional classes without objects in the instance model.

The most crucial consequence of these definitions is that the handling of refinement (inheritance and instance-of) relations is identical for basic entities, connections and mappings (while there is a certain orthogonality for compound entities/models). As a result, **two model elements can simultaneously be in subtype and instance-of relations**, which is a major difference in contrast to the MOF standard.

In order to obtain a complete definition of our refinement calculus (that encapsulates the metamodel of Fig. 2.11 within our framework), a top section of the inheritance and containment hierarchy is introduced as follows.

**Definition 2.13 (Model space of VPM).**

The model space of the VPM framework always contains (at least) the following elements.

- The abstract model element *Universe* or *Top* is greater than all the other model elements (entities, connections and mappings), thus being the root of both the inheritance and the containment hierarchy.
- The entity *Entity* is contained by (and refined from) *Universe*.
- The connection *Connection* is leading from and to *Entity*.
- The mapping *Mapping* is leading from and to *Entity*.

Thereafter, any well-formed model space has to fulfill the following axioms.

**Property 2.14 (Inheritance and instantiation is partial order).** Each element in the model space (except for *Universe*) has at least one supertype, and both refinement relations (inheritance and instantiation) are reflexive, transitive and anti-symmetric.
Informally, multiple inheritance and instantiation are allowed but circularities are therefore forbidden in the type hierarchy (to be precise, treated as equality). Note that the definition formally permits that an element is inherited from (alternatively, instance of) itself, which only eases the mathematical treatment of our framework without foregoing consequences on the intuitive meaning.

Property 2.15 (All model elements are contained). Each element in the model space (except for Universe) is contained by at least one element, moreover, the containment relation is transitive.

As a consequence, multiple and circular containment are thus allowed by this axiom, and each element should be reachable from the top element by navigating containment relations.

2.4.3 Formalizing the Petri net metamodel hierarchy

The theoretic aspects of model refinement (and instantiation) are now demonstrated on the Petri net metamodel hierarchy. Supposing that the refinement relations depicted at the bottom of Fig. 2.10 hold between the model elements (e.g., Boy is a refinement of Node, el is a refinement of tokens; the interested reader can verify that all the connection refinements are valid) we can observe the following.

Proposition 2.16. BipartiteGraph is both a(n entity) subtype and instance of Graph.

Proof. The proof consist of two steps.

1. **Proof of refinement:** for each element in the Graph model there exists a refinement in BipartiteGraph. 
   Girl is refinement of Node; GBEEdge is of Edge; from1 is derived from from and to is refined from to.

2. **Proof of instantiation:** for each element in BipartiteGraph there exists an instance-of relation in Graph. Girl and Boy are instantiations of Node; GBEEdge and BGEEdge are of Edge; from1 and from2 are of from; and to1 and to2 are of to. □

By similar course of reasoning, we can prove all the other relations between different models of Fig. 2.10. Note that Petri Net is not an instance of Bipartite Graph (as Token is a new element in PetriNet), and SimpleNet is not inherited from Petri Net (since, for instance, there are no transitions in SimpleNet).

2.4.4 Pattern refinement

We introduce the notion of pattern refinement as a special case of entity refinement, which provides a means to formally capture the use of design patterns (i.e., how an abstract pattern can be embedded in a concrete user model) by entity refinement. Moreover, pattern refinement will also form the bases of rule refinement later in Sec. 4.5.

The overall idea (and typical use) of pattern refinement is depicted in Fig. 2.12 where SubPattern is intended to be a refinement (⊆) of SuperPattern.

We suppose that the abstract pattern SuperPattern stored in a pattern library is a compound entity containing a “metamodel” entity SuperMeta specifying type information and a model entity SuperModel (which is an instance of SuperMeta) describing the designated use of the pattern. Thereafter, in a concrete user model (SubModel in package SubPattern) aiming to apply the pattern properly in a application domain defined by the metamodel SubMeta, one has to establish the inheritance relation between SuperPattern and SubPattern by showing that SuperMeta is a generalization of SubMeta (i.e., the application domain is a proper refinement of the pattern metamodel) and SuperModel is a generalization of SubModel (i.e., the user model contains at least the elements required by the library pattern SuperModel).

As a summary, we can formally define patterns and pattern refinement as follows.

**Definition 2.17 (Pattern).** A (MOF) pattern $P$ is a compound entity consisting of entities $Meta$ and $Model$ where $Model \rightarrow Meta$. A VPM (graph) pattern is simply an entity $P$. 
Definition 2.18 ((VPM) Pattern refinement). A pattern $P_{\text{sub}}$ is a pattern refinement of $P_{\text{super}}$, if $P_{\text{sub}} \subseteq P_{\text{super}}$.

Definition 2.19 (Constants and variables in patterns). Let $P_{\text{sub}} \subseteq P_{\text{super}}$ be a pattern refinement. If $\exists i, j : (X = P_{\text{super}}[i] \wedge X = P_{\text{sub}}[j])$ then $X$ is a constant in pattern $P_{\text{super}}$ (when matching to $P_{\text{sub}}$) otherwise $X$ is a variable. The variables of a pattern are denoted as $\text{var}(P) = \overline{X} = \langle X_1, \ldots X_n \rangle$.

Constants and variables are distinguished from each other using the well-known Prolog convention, i.e., the names of variables are started with capital initials (and constants are well-formed Prolog atoms). Typically, all elements in the Meta component of a pattern are constants (i.e., identifiers of metamodel classes), while the elements in the Model component are either constants or variables.

2.4.5 The four-layer MOF architecture in VPM

A main advantage of our approach (in contrast to e.g., [7] or the MOF standard itself) is that type-instance relations can be reconfigured dynamically. On one hand, as a model can take the role of a metamodel (thus being simultaneously a model and a metamodel) by altering only the single instance relation between the models, we avoid all the problems of Sec. 2.3. On the other hand, transformations on different metalevels can be captured uniformly, which is an extremely important feature when considering the evolution of models through different domains.

Furthermore, our metamodeling framework clearly demonstrates that the fixed number of metalevels introduced by the MOF standard is artificial and mathematically unsound: the four layers are finite restrictions of a general refinement relation, which is transitive both in case of inheritance and instantiation.

On the other hand, as VPM is more general than MOF, the original four-layer MOF architecture can be embedded into VPM by introducing the following constraints (in fact, they can be captured in a constraint language like OCL as static well-formedness rules).

1. The metametamodel defined by MOF can be introduced as a compound entity refined from our top-level concepts.
2. The metamodels of modeling languages (like the UML metamodel, Petri Net metamodel etc.) are entity instances of the MOF metamodel entity.
3. User models can be instances of the metamodel entities.
4. The object level instances have in turn related entities to corresponding user models.
5. Within the same “metalevel”, arbitrarily long chains of inheritance relations are allowed.
2.5 Static Consistency Analysis of Metamodels

Although, in the Sec. 2.4, the concepts of models and metamodels have been formalized precisely, in the sequel, we introduce an equivalent representation (called refinement graphs) to visualize and automatically detect flaws in the refinement hierarchy. Refinement graphs eliminate the distinction between entities, connections and mappings, however, all type information is preserved. After that, we show (i) how one can judge whether a certain model is (in a consistent way) more abstract or more refined than another one, and (ii) how certain inconsistencies in merging models (packages) can be corrected automatically based directly on the abstract representation.

The practical feasibility of our approach is demonstrated on formalizing advanced concepts of metamodeling including structural extensions, type restrictions and recent concepts from the new UML 2.0 standard [162] (like import, redefine and package merge constructs).

2.5.1 Refinement graphs

Definition 2.20 (Refinement graph). The refinement graph $RG = (Nodes, Edges)$ of a given model space is a directed graph defined as follows.

- A refinement node $n \in Nodes$ is either an entity, a connection, or a mapping.
- A refinement edge $e \in Edges$ (that can be interpreted informally as implication) leads from node $n_1$ to node $n_2$ (denoted as $n_1 \xrightarrow{e} n_2$, or simply $n_1 \rightarrow n_2$) when
  1. Inheritance: for the corresponding model elements of nodes $n_1$ and $n_2$, the model element of $n_1$ is (directly or indirectly) inherited from the model element related to $n_2$; formally, $n_1 \rightarrow n_2$
  2. Instantiation: for the corresponding model elements of nodes $n_1$ and $n_2$, $n_1$ is (directly or indirectly) an instance of $n_2$; formally, $n_1 \rightarrow n_2$
  3. Source of Connection/Mapping: $n_1$ is related to a connection (mapping) leading from the entity itself associated to $n_2$ or one of its subentities;
  4. Target of Connection/Mapping: $n_1$ is related to a connection (mapping) leading to the entity itself associated to $n_2$ or one of its subentities;
- Extensions: For the mathematical treatment, let $n_\bot$ be a node having only outgoing edges that is linked to all the other nodes, and let $n_\top$ be a node (corresponding to Universe) having only incoming edges leading from all other nodes.

Informally, an entity node has an outgoing edge to all its “super” entity nodes (by merging the concepts of inheritance and instance of relations), while a connection (mapping) node has an outgoing edge to all its “super” connections (mappings) edges, plus all the “super”entities of its source and target entity node.

Example 2.21. In Fig. 2.13, the refinement graph of simple model space containing the concepts of State and Transition of a finite automaton can be observed. For the sake of clarity, the model space is incomplete (not well-formed) in the sense that containment relations are not depicted.

The figure in the middle explicitly depicts all the edges of the refinement graph (except for self loops, and extension edges leading from $n_\bot$ and to $n_\top$). The figure in the right depicts (which will be our standard notation for the rest of the paper to improve clarity) only the direct refinement edges. The entire set of edges can be calculated by the transitive (and reflexive) closure of the explicitly depicted edges.

Moreover, the edges with a white arrowhead (like UML generalizations) will refer to inheritance and instantiation relations in the model (instantiation edges are labeled with $inst$), while edges with ordinary arrowhead (like navigable associations in UML) denote source and target restrictions (labeled with $src$ and $trg$, respectively) for connections and mappings.

According to a final notational convention, nodes derived from entities appear in black, nodes related to connections in white, while nodes of mappings have a striped background.
A refinement graph of a model explicitly depicts all the type constraints expressed by the metamodels and models in the sense that whenever an edge is leading from a node $n_1$ to node $n_2$ we can deduce that $n_2$ is more abstract than $n_1$. In other words, the definition of model element $n_2$ must exist prior to introducing model element $n_1$.

### 2.5.2 The lattice of a refinement graph

In the following, we formally introduce two lattices representing (i) a single refinement graph and (ii) the set of refinement graphs to provide a formal analysis mechanism for type compliant evolution of models, e.g., to decide whether (i) a specific model is well-typed, (ii) and a model is a proper refinement of another (more abstract) model. Moreover, lower and upper bound operations will provide means to integrate different versions or different aspects of a model (e.g., created by different designers) into a safe and consistent global viewpoint of the system that is required by the advanced package merge constructs in the UML 2.0 standard.

**Definition 2.22.** A lattice $L(N) = (\sqsubseteq_N, \sqsubseteq_N, \sqcap_N, \sqcup_N, \sqcap_N)$ is a five tuple where

1. $\sqsubseteq_N$ is a partial order on the set $N$,
2. $\sqsubseteq_N$ is the infimum of $N$, that is $\forall n \in N \sqsubseteq_N \sqsubseteq n$
3. $\sqcap_N$ is the supremum of $N$, i.e., $\forall n \in N n \sqsubseteq_N \sqcap_N$
4. $\sqcup_N$ is the least upper bound of a set $N_1 \subseteq N$ is defined as $\sqcup_N N_1 = u \iff \forall n \in N_1 : n \sqsubseteq u \land \forall u' \in N : n \sqsubseteq u' \supset u \sqsubseteq u'$.
5. $\sqcap_N$ is the greatest lower bound of a set $N_1 \subseteq N$ is defined as $\sqcap_N N_1 = u \iff \forall n \in N_1 : u \sqsubseteq n \land \forall u' \in N : u' \sqsupset n \supset u' \sqsubseteq u$.

The least upper bound $\sqcup_N$ of a set $N_1 \subseteq N$ is the least element that is larger than any elements in $N_1$. This can be determined by, for instance, a reachability analysis: (i) initially, the $\sqcap_N$ element is added as singleton to the current set as it is an upper bound of all elements $n \in N_1$; (ii) all the predecessors of elements in the current set are tested whether they are still an upper bound of $N_1$. Those that satisfy this condition become the next current set, and this process is iterated until a fixpoint is reached.

The calculation of the greatest lower bound $\sqcap_N$ is similar, but this time one should start from the bottom element $\sqsubseteq_N$ element and perform the dual steps.

As a direct consequence of its definition, one can show that the nodes of refinement graphs form a lattice.
Proposition 2.23. The nodes of the refinement graph \( RG = (\text{Nodes}, \text{Edges}) \) form a lattice \( L(\text{Nodes}) = (\subseteq, \bot, \top, \cup, \cap) \) with \( \subseteq = \text{Edges} \) (the partial order of nodes is imposed by the edges), \( \bot = n_\bot \) (the bottom extension node as infimum), \( \top = n_\top \) (the top extension node as supremum), while \( \cup \) and \( \cap \) are defined as usual according to \( \subseteq \).

As a result, many type conformance questions of a single model (like the following ones) can be answered directly on refinement graphs by the least upper bound and greatest lower bound operations of the lattice.

- Is a model element \( A \) a supertype of model element \( B? \) \( \equiv \) the least upper bound of \( A \) and \( B \) is equal to \( B \).
- What is the common supertype of a set of model elements? \( \equiv \) the least upper bound of the set.
- Does a link (model-level connection) correspond to its association (meta-level connection)? \( \equiv \) the least upper bound of link connection \( L \), its source object entity \( O_{src}\), the association connection \( A_{sup} \) of the link and the source class entity \( C_{src} \) is equal to \( C_{src} \) (due to the influence of instantiation and connection end edges in the refinement graph);
- Is there such a model element that is inherited (instantiated) from both model element \( A \) and \( B? \) \( \equiv \) the greatest lower bound of \( A \) and \( B \) is not equal to \( \bot \).

Surprisingly, even more interesting results can be obtained if we establish another lattice for a set of refinement graphs.

2.5.3 The lattice of the sets of refinement graphs

In order to show that the set of refinement graphs also forms a lattice, we need to establish the notions of (1) the partial order relation on the set, (2) the infimum, and (3) the supremum of the set, and finally, (4) the least upper bound (hub) and (5) greatest lower bound (glb) operations.

Proposition 2.24. Let \( \text{Set}(RG) \) be the (finite) set of all finite refinement graphs (relevant for a specific purpose). The traditional subgraph relation is a partial order \( \subseteq \) on the set \( \text{Set}(RG) \) of refinement graphs.

This is a well-known result from elementary graph theory, therefore not proven here.

This time the calculation of greatest lower bound and least upper bound of a subset \( X \subseteq \text{Set}(RG) \) needs some precautions in order to maintain the property that they both yield a well-formed refinement graph as the result.

Greatest lower bound

The calculation of the greatest lower bound \( \cap X \) of a subset \( X \subseteq \text{Set}(RG) \) takes the intersection of all the refinement graphs in \( X \), and the result is expected to be a common consistent abstract model of all the models in \( X \). For that reason, at most those nodes and edges are included in the result lattice that appear in all \( x \in X \) lattices.

However, different models that share the same nodes may be in a conflict. Suppose that there is a refinement graph \( M_1 \) where \( n_1 \rightarrow n_2 \) but in another model \( M_2 \), it is just the opposite \( n_2 \rightarrow n_1 \). Since \( \rightarrow \) is a partial order, \( n_1 \rightarrow n_2 \wedge n_2 \rightarrow n_1 \supset n_1 = n_2 \), which also fulfills our expectations, as both configurations can be refined from an abstract model where the two nodes have not been distinguished yet. As there are no other refined models where these conditions could also be satisfied, we showed that by this method exactly the greatest lower bound of \( X \) is derived.

Note that the refinement graph consisting of the single \( n_\top \) node is the infimum of the entire set \( \text{Set}(RG) \). Thus we have the following proposition.
Proposition 2.25. The set Set(RG) of refinement graphs has an infimum \( \bot_G \) (which is the graph consisting of the single node \( n_\top \)), and a greatest lower bound \( \cap_G X = RG(\text{Nodes}, Edges) \) such that
1. \( \forall n \in \text{Nodes} : n \in x_1.\text{Nodes} \land \ldots \land x_n.\text{Nodes}, x_1, \ldots, x_n \in X \) (all the common nodes in the refinement graphs \( x_i \) are included)
2. \( \forall n_1, n_2 \in \text{Nodes} : n_1 \rightarrow n_2 \land n_2 \rightarrow n_1 \equiv n_1 = n_2 \) (conflicting nodes are merged into a single node as a consequence of \( \rightarrow \) being a partial order)
3. \( \forall e \in \text{Edges} : e \in x_1.\text{Edges} \land \ldots \land x_n.\text{Edges}, x_1, \ldots, x_n \in X \land e \in \text{Edges} \land e.\text{to} \in \text{Nodes} \) (all the common edges are added that have both source and target nodes in \( \text{Nodes} \))

Corollary 2.26. For all \( G_1, G_2 \in \text{Set}(MG) : \cap_G \{ G_1, G_2 \} \subseteq G_1 \) and \( \cap_G \{ G_1, G_2 \} \subseteq G_2 \).

Informally, the greatest lower bound of two refinement graphs is a refinement of both of them.

Least upper bound

When calculating the least upper bound of some \( X \subseteq \text{Set}(RG) \), one has to take the union of all the graphs \( x \in X \), and, according to our informal expectations, this union should be a refinement of all individual model graphs. Unfortunately, in the case of contradicting models, this property cannot be established.

When taking the union of graphs \( x \in X \), a merging is required along the nodes that appear in more than a single model. After that, the calculation of the union of the edges adds an edge leading from node \( n_i \) to node \( n_j \) if there is at least one model \( x_k \) with such an edge but there are no models with an edge leading to the opposite direction (i.e., from \( n_j \) to \( n_i \)). In the latter case, no edges are established in the result refinement graph between \( n_i \) and \( n_j \) in accordance with the properties of implication \( n_i \rightarrow n_j \lor n_j \rightarrow n_i = \top \).

For the practical use, when a least upper bound of the set \( X \) is not a refinement of one or more models, the user can automatically return to the greatest consistent global state in the past by taking the greatest lower bound of \( X \), which is inevitably a proper abstraction, and the refinement process can be redone from there in a controlled way.

Proposition 2.27. The set \( \text{Set}(RG) \) of refinement graphs has a least upper bound \( \cup_G X = M(\text{Nodes}, Edges) \) such that
1. \( \forall n \in \text{Nodes}, x_1, \ldots, x_n \in X : n \in x_1.\text{Nodes} \lor \ldots \lor x_n.\text{Nodes} \) (all the common nodes)
2. \( \forall e \in \text{Edges}, x_1, \ldots, x_n \in X : e \in x_1.\text{Edges} \lor \ldots \lor x_n.\text{Edges} \land e.\text{from} \in \text{Nodes} \land e.\text{to} \in \text{Nodes} \) (all the common edges)
3. \( \forall n_1, n_2 \in \text{Nodes} : \neg(n_1 \rightarrow n_2 \land n_2 \rightarrow n_1) \) (conflicting edges are removed as a consequence of \( \rightarrow \) being a partial order)

If the set \( \text{Set}(RG) \) of refinement graphs is not controversial then we have a supremum \( \top_G = \cup_G \text{Set}(RG) \), which is the least upper bound of all graphs in \( \text{Set}(RG) \).

Example 2.28. In Fig. 2.14 the greatest lower bound and least upper bound of a set consisting of two models (introducing accepting and initial states in a deliberately contradicting way for finite automata) is calculated. For better understanding, the visual representations of the models are also depicted.

- In the case of the least upper bound (lub), the examination of the refinement graph detects that nodes \( \text{Acc} \) and \( \text{Init} \) are contradicting. For this reason, the lub supposes that both were introduced on purpose and keeps both of them, but the ordering relation between them is removed. As a result, we have an \( \text{AccState} \) and \( \text{InitState} \) derived by object inheritance from \( \text{State} \), which is not a refinement of the two models (due to the contradicting inheritance).
• In the case of the greatest lower bound (glb), the examination of the refinement graph detects that nodes Acc and Init are equal. For this reason, they are treated as if all the edges entering one of them ends in this common state. As a result, we have an AccState derived by object inheritance from State but InitState must be reintroduced later in the design (or vice versa).

As a consequence, we can show that the sets of refinement graphs also form a lattice.

**Proposition 2.29.** The set Set(RG) of refinement graphs (ordered by the subgraph relation) forms a lattice \( L_G = (\subseteq_G, \bot_G, \top_G, \cup_G, \cap_G) \).

Although we showed that the refinement graph structure of a single model is preserved when calculating lower and upper bounds of sets of refinement graphs, the result might be unexpected at the diagram level. For instance, if a connection (mapping) is redirected from one entity to another entity, the intersection of the two models might contain only the edge between the source entity and the connection (mapping) while the target of the connection (mapping) remains unspecified. However, this situation can easily be detected by comparing the degrees of each connection (mapping) node with the ones in the original refinement graph. Similarly, when taking the union of model representation, a connection (mapping) node may have additional outgoing edges, which fact can be detected similarly within the semantic domain. Both modeling contradictions are resolved automatically by our technique to a common consistent view as much as (theoretically) possible.
2.5.4 Practical uses of refinement graphs

In Fig. 2.15, we demonstrate how some major concepts of object-oriented metamodelling can be captured formally by refinement graphs on our running examples of finite automata.

![Refinement Graphs](image)

Fig. 2.15. Structural extension, relation restriction, instantiation

**Structural extension**

In a structural extension, a new model element is added to the model by refining (by inheritance or instantiation) an existing model element following our axioms.

For instance, a new entity (class) AccState has been derived by inheritance to the original diagram of Fig. 2.13 from the existing entity State. Although this time the designer can be quite certain that the refinement step he or she performed is correct, this can be verified formally on the semantic level by comparing the new refinement graph Model₁ to Model₀ of Fig 2.13. One can conclude that the refinement step is correct as Model₀ ⊆ G Model₁ because node Acc and its incoming and outgoing edges were introduced by the refining operation.

Up to this point, the diagrams contained only the relationship on the class level. Now, when deriving Model₃ from Model₁ (and, similarly, Model₄ from Model₂), an instance S1 of class AccState is introduced by refinement. This example demonstrates that classes and instances can be treated uniformly by the refinement graph: an instance is another node of the refinement graph derived from its class by instantiation. In many cases, both instantiation and inheritance holds between two entities (connections, mappings), which stems from the fact, mathematically, that a singleton (new element introduced as a leaf) only contains its own identifier.
Moreover, altering a model by the following set of operations turns out to be formally a model refinement in our sense (not proven here).

- Inserting an entity (e.g., a UML class) consistently as a leaf element or between two existing entities (refining the entity inheritance tree).
- Inserting an entity (e.g., a UML object) consistently as a leaf element into the instantiation hierarchy.
- Inserting a connection (e.g., a UML association or link) consistently (i.e., maintaining type constraints) as a leaf element or between two existing connection records (refining the connection inheritance tree).
- Inserting a mapping (e.g., a UML attribute or slot) consistently (i.e., maintaining type constraints) as a leaf element or between two existing connection records (refining the connection inheritance tree).
- Introducing multiple inheritance (or instantiation) for entities, connections or mappings (in which case the inheritance structure is no longer a tree but a directed acyclic graph).

On the one hand, several incorrectness properties (such as circularity in the inheritance structure, or invalid type refinements) can be detected on the refinement graph itself. In fact, they will never be introduced if the model refinement process is guided by the $\text{glb} \cap G$ and $\text{lub} \cap G$ operations.

On the other hand, the uniform refinement graph structure also allows a particular metamodeling approach to distinguish between classes and instances by introducing instantiation as a connection and thus deriving instances by connection instantiation.

**Type restriction**

In a type restriction, our model is not extended but altered by redirecting the refinement of an entity, connection or mapping. This could possibly mean that

- An entity refinement (either inheritance or instantiation) relation is established between two entities having previously the same parent in the inheritance structure. However, note that the inheritance of connections and mappings cannot always be redirected in this way as the proper inheritance of sources and targets may not be assured.
- The source (target) of a connection (or mapping) is redirected to a subtype of its former source (target) entity.

Our running example covers the latter case when $Model_2$ is generated from $Model_1$ (and, additionally, when $Model_4$ is derived from $Model_3$). In Fig. 2.15, the dashed lines can be derived by the transitive closure of solid lines; however, they are depicted explicitly to improve clarity of the subgraph relation when verifying that $Model_2$ is a proper refinement of $Model_1$.

**“Future” concepts: redefine, import, package merge**

The metamodeling concepts of the new kernel of the UML 2.0 standard [162] are based upon three constructs (represented by dependencies with corresponding stereotypes) interpreted on packages (i.e., thus handling models and metamodels), namely, redefine, import, and package merge.

By the redefine operation (already well-known from many object-oriented programming languages that allow overriding of methods and properties), one can introduce a class derived from an existing one by inheritance with identical names but altered content in a new context (package). In a strict mathematical sense, the result of redefine should be a proper refinement of the redefined class. Therefore, as our VPM framework (and their refinement graph representation) does not depend on names, we can simply say that a redefine operation (if consistent) introduces new refinement (inheritance, to be precise) relations into the model space with identical names.
The import construct (that always leads between two packages) prescribes that all the contents of the source package are implicitly included in the target package. In our framework, the import construct is redundant in the sense that (i) on the one hand, a model element can be contained by multiple packages, (ii) on the other hand, refinement relations can “cross the borders of a package”, thus there is no need to include them first in order to capture the intended refinement.

The package merge construct (demonstrated in Fig. 2.16) is probably the most complex metamodeling concept in UML 2.0.

![Fig. 2.16. Package merge](image)

The overall idea is to derive metamodels (Package C in the figure) by merging existing ones (Package A and B) using the redefine and import operations in such a way that all the original associations and generalizations are also included in the result for the target package (Package Result).

- If two model elements with identical names appear both in the target and in a source package (for instance, see State in Package A and C), the package merge construct introduces a new generalization, and the class in the target package (C.State) is inherited from the class in the source (A.State).
- If such a model element is encountered that only appears in (at least) one of the source package but not in the target package (like the from connection in Package A or B but not in Package C), the model element is simply added.

However, the UML proposal [162] does not specify what happens (in the very common case) if the merge of multiple source packages results in inconsistencies (like the unclear intension of generalization between AccState and InitState in Fig. 2.16).

From a VPM point of view, package merges can be specified as multiple entity inheritance (in case of compound entities) between the source entities and the target entity. As a result of our static consistency analysis technique based upon the lub and glb calculation for refinement graphs, we can
pinpoint such inconsistencies during package merge, furthermore, a consistent viewpoint (i.e., consistent from a refinement point of view and not from the viewpoint of package merge) is automatically obtained (as done in Fig. 2.14).

2.6 Conclusions

I presented a visual, and formally precise metamodeling (VPM) framework that avoids the problems of MOF metamodeling by uniformly handling arbitrary metamodels and models taken from both engineering and mathematical domains. This uniform treatment is achieved by generalizing the traditional inheritance and instantiation relations and formalizing the structure of traditional mathematical definitions.

- **Separation of static and dynamic model elements.** In addition to static elements of metamodels I introduced the well-separated concept of dynamic model elements (Sec. 2.2.1 based upon [168,172, 181,186]).

- **Structural refinement of models.** Based upon a minimal core subset of MOF elements (classes / entities, associations / connections, attributes / mappings), I elaborated the refinement calculus of structural model elements (Sec. 2.4 following [181,186]). Unlike the MOF and UML standards, this simple refinement calculus includes the inheritance and instantiation of all model elements (associations, attributes and packages) in addition to the refinement of classes. It is worth emphasizing that these refinement relations can be reconfigured dynamically during the evolution of VPM models.

- **Static consistency analysis.** I proposed a general method (Sec. 2.5 based on [179,186]) based upon refinement graphs (which visualizes the partial order relation imposed by inheritance and instantiation relations) that allows to formally detect and automatically resolve models and metamodels violating the axioms of our static refinement calculus.

**Conceptual relevance**

The main conceptual advantage of the VPM metamodeling framework is that it provides deeper modeling facilities than the MOF standard, however, it is based upon a much more concise kernel language. I extend MOF in depth and not in width (as done in the new UML 2.0). This statement is demonstrated on a VPM benchmark aiming at a quantitative analysis of UML models [134,186].

Another conceptual advance will be revealed after the upcoming chapter when the dynamic behavior of a language will be formally captured by graph transformation rules. Since the metamodel of rules can be constructed [189] as well, they can be stored and handled as traditional models, and only the interpreter has to distinguish between operation and data. In this respect, we lifted the *Neumann principle* from computer architectures to modeling frameworks. The main idea in achieving this lies in the explicit modeling of the instance-of relation (in addition to the traditional inheritance relation) and the uniform (non-separated) treatment of classes and objects.

Furthermore, the separation of static and dynamic elements (which is missing from the standard) is the first step towards formalizing the dynamic behavior of modeling languages in a visual way prior to applying graph transformation techniques. This separation will also play a crucial role in Chapter 6 to synthesize an efficient model checking input description for the verification of modeling languages and transformations.

Finally, the static consistency analysis technique provides a prime mathematical basis for appropriately handling the advanced modeling concepts (like package merges) of the new UML 2.0 kernel language.
Practical relevance

A main practical relevance of the VPM metamodelling framework is that it allows the construction of UML and Meta-CASE tools of much higher quality (due to the formal mathematical background) and flexibility (due to our fluid meta-level concept). Meta-CASE tools (such as Meta-Edit [110]) can be tailored to the special needs of specific application domains than UML. Moreover, with the dynamic behavior introduced in the upcoming chapter, the performance and modeling capabilities of such tools would highly exceed the state-of-the-art Meta-CASE tools.

Also, the use of metamodelling techniques for the development of a complex mathematical analysis tool (aiming to carry out dependability analysis of BPM-based e-business applications with dataflow networks) have been found very useful in [50].

Furthermore, since the VPM core is very close to the structure of XML documents, it is easy to adapt the entire formalism to a model-driven transformation framework for XML documents. In this view, the metamodels of the source and target XML document are constructed based upon the XML Schema or the DTD descriptions. The mapping between XML documents and VPM models can easily be automated. As a result, the designers have a very high-level visual alternative for XSLT [159] at hand, which can be especially useful for constructing complex semantic XML-to-XML or XML-to-code transformations (e.g., generating a Java program from the XMI representation of a UML statechart or an Action Semantics expression) where XSLT typically fails to scale up. In fact, the VIATRA tool already implemented this approach to carry out transformations between documents following the XMI standard [121].

The static consistency analysis technique (of Sec. 2.5) could play a central role in view integration which is a central problem, for instance, in aspect-oriented programming [97]. Traditional solutions to the view integration problem typically require that the separate views should be consistent otherwise they are unable to merge them into a consistent global view of the system. My technique yields a partial result for inconsistent views as well, and it is also able to highlight inconsistent parts in the local views.

Out of scope: well-formedness constraints

Note, however, that even though our metamodelling framework is much more concise (concerning the number of elements introduced as the kernel) and expressive (with dynamically reconfigurable metalevels), a metamodel alone cannot capture all the static well-formedness constraints of a modeling language. In the chapter, we decided not to fix the way how domain engineers can specify additional static restrictions. On one hand, such constraints can be expressed by using OCL, in which sense our methodology is complimentary to existing techniques. On the other hand, a graph transformation rule without side effects (i.e., with identical left-hand side and right-hand side) can be interpreted as a static graphical constraint [169], in which case our rule refinement method provides in turn a certain level of reusability.

Future work

Future work should primarily focus on the adaptation of VPM metamodelling techniques in a mathematical context. Further research should also aim at providing an automated translation of mathematical structures (from formal definitions given e.g., in a MathML format) into their corresponding VPM metamodel in order to ease the handling of mathematical domains for transformation engineers.
VPM Models as Abstract State Machines

Based upon the definitions of Section 2.4.2, we introduce an algebraic representation of VPM modeling concepts by Abstract State Machines (ASMs) that will serve as the basis of relating different semantic frameworks later in the thesis. Moreover, we define elementary operations that provide a consistent manipulation of VPM models.

3.1 Motivation: Elementary VPM operations on a uniform semantic representation

In the previous chapter, we defined an axiomatic semantics for VPM models. This provides a satisfactory level of preciseness from a theoretical aspect; however, axioms alone do not show guidelines how to actually maintain the consistency of the model space. Thus from a practical point of view, our results give little help for tool builders who aim at implementing the VPM metamodeling framework.

Problem statement

Therefore, a set of basic operations is necessitated which consistently manipulate VPM models. In other terms, if we have a consistent VPM model and we execute any of these elementary operations, the result is also a consistent VPM model.

Moreover, these operations have to be specified in a formal (operational or denotational) way using an easy-to-understand semantic framework, which is as universal as possible in order to ease the implementation of the VPM approach in various programming languages and architectures.

Related work

Abstract state machines [74] have become a popular and universal formalism for capturing the semantics of various programming languages and modeling techniques at the proper level of abstraction. The universality of ASMs is demonstrated in [75] where Gurevich claims that each algorithm can be formally captured by an appropriate (sequential) abstract state machine.

ASMs have been successfully applied to describe the precise mathematical foundations based on the step-wise refinement of models for many standards including the virtual machine (Warren Abstract Machine) of Prolog [32], Java [33], UML [27,28,115], the IEEE standard for VHDL’93 [29], the ITU standard for SDL’2000 [92] and many more.

ASMs has been connected to various back-end formal verification tools in several case studies. The use of ASMs for proving the correctness of compilation schemes by theorem proving techniques has been researched in a part of the Verifix project [72]. Mapping ASM descriptions into model checkers has been studied in [196].

In addition to the theoretical well-foundedness of ASMs, there is also industrial interest, for instance, at Microsoft where the AsmL language frequently serves as a common formalism for the
specification, simulation and testing of component-based software development [20, 21]. The AsmL specification language runs on the .Net platform and exploits the abstraction potential of ASMs to offer component-based and object-oriented structuring principles.

The wide range of industrial application also cover, for instance, the specification of several processors and computer architectures (e.g., the ARM2 advanced RISC machine [89]), cryptographic protocols (e.g., the Needham-Schroeder protocol in [22]), database transactions [138], or testing [69].

However, up to our knowledge, ASMs have not yet been applied in formalizing metamodeling frameworks and the structure of and visual modeling languages, which is the primary objective for the current chapter. Due to the recent ASM support at Microsoft Research, our theoretical foundations of VPM metamodeling can easily be integrated into off-the-shelf programming languages and tools.

**Objectives**

Our objectives in the current chapter are the following.

1. A brief overview is provided first (in Sec. 3.2) on the basic definitions of abstract state machines.
2. Then, in Sec. 3.3 we propose a meta-level algebraic representation of VPM models) based on ASMs.
3. This also includes several well-formedness constraints (in the form of invariants; Sec. 3.3.2) that aim to maintain the consistency of the algebraic VPM representation with respect to the original VPM construction discussed in Sec. 2.4.
4. Section 3.4 proposes elementary VPM operations that consistently manipulate the algebraic representation.
5. Then, in Sec. 3.5, we reintroduce the algebraic representation (and elementary operations) now on the model-level in order to obtain the traditional algebraic representation of MOF based models.
6. Finally, Sec. 3.6 concludes the chapter with summarizing its theoretical and practical relevance.

![Diagram](image)

**Fig. 3.1. Information flow of Chapter 3**

The major information flow summarizing the organization of sections in the current chapter is depicted in Fig. 3.1. Section numbers appearing in a circle are cross-references to sections already discussed which have a major impact on certain parts in the current chapter. Sections with a special emphasis for reading are highlighted again.
3.2 An Introduction to Abstract State Machines (ASMs)

Our summary below on abstract state machines is essentially based upon [30, 33].

3.2.1 Vocabulary and states of ASMs

In a state of our model space, data is represented as abstract elements of domains (also called universes, one for each category of data) which are equipped with basic operations as functions. Without loss of generality we treat relations as boolean valued functions and view domains as characteristic functions, defined on the superuniverse which represents the union of all domains. Thus the states of our model space are algebraic structures (called simply as algebras).

Definition 3.1 (Vocabulary). A vocabulary \( \Sigma \) is a finite collection of function names. Each function name \( f \) has an \( \text{arity} \), a non-negative integer, which is the number of arguments the function takes. Function names can be \( \text{static} \) or \( \text{dynamic} \).

Nullary function names are often called \( \text{constants} \); however, this term is misleading as the interpretation of dynamic nullary functions can change in ASMs so that they correspond to variables of programming.

Example 3.2. For instance, the vocabulary \( \Sigma_{\text{Bool}} \) of Boolean algebras contains two constants 0 and 1, a unary function name \( '-' \), and two binary function names \( '+' \) and \( '*' \).

Each consistent configuration of the VPM model space will constitute a \( \text{state} \) of an ASM.

Definition 3.3 (State). A state \( \mathfrak{A} \) of the vocabulary \( \Sigma \) is a non-empty set \( X \) (the \text{superuniverse} of \( \mathfrak{A} \), denoted as \( |\mathfrak{A}| \)) together with interpretations of the function names of \( \Sigma \).

- If \( f \) is an \( n \)-ary function name of \( \Sigma \), then its interpretation \( f^\mathfrak{A} \) is a function from \( X^n \) into \( X \).
- If \( c \) is a constant of \( \Sigma \) then its interpretation \( c^\mathfrak{A} \) is an element of \( X \).

Example 3.4. We may define a state \( \mathfrak{A} \) for the vocabulary \( \Sigma_{\text{Bool}} \) as follows. The superuniverse of the state \( \mathfrak{A} \) is the set \( \{0, 1\} \). The functions are interpreted as follow, where \( a \) and \( b \) are 0 or 1.

\[
\begin{align*}
0^\mathfrak{A} & := 0 \quad \text{(zero)} \\
1^\mathfrak{A} & := 1 \quad \text{(one)} \\
\overline{\mathfrak{A}}a & := 1 - a \quad \text{(logical complement)} \\
a +^\mathfrak{A} b & := \max(a, b) \quad \text{(logical OR)} \\
a \times^\mathfrak{A} b & := \min(a, b) \quad \text{(logical AND)}
\end{align*}
\]

Formally, function names are interpreted in states as total functions. However, we may view them as being partial and define the \text{domain} of an \( n \)-ary function name \( f \) in \( \mathfrak{A} \) to be the set of all \( n \)-tuples \((a_1, \ldots, a_n) \in |\mathfrak{A}|^n \) such that \( f^\mathfrak{A}(a_1, \ldots, a_n) \neq \text{undef} \).

The constant \( \text{undef} \) represents an undetermined object, the default value of the superuniverse. It is also used to model heterogeneous domains. In applications, the superuniverse of a state \( \mathfrak{A} \) is usually divided into smaller \text{universes}, modeled by their characteristic functions. The universe represented by \( f \) is the set of all elements \( t \) for which \( f(t) \neq \text{undef} \). If a unary function \( f \) represents a universe, then we simply write \( t \notin f \) as an abbreviation for the formula \( f(t) \neq \text{undef} \).
3.2.2 Terms, variable assignment and formulae

**Definition 3.5 (Term).** The terms of $\Sigma$ are syntactic expressions generated inductively as follows:

1. Variables $v_0, v_1, v_2, \ldots$ are terms.
2. Constants $c$ of $\Sigma$ are terms.
3. If $f$ is an $n$-ary function name of $\Sigma$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

Terms are denoted by $r, s, t$; variables are denoted by $x, y, z$. A term which does not contain variables is called closed.

**Example 3.6.** The following are terms of the vocabulary $\Sigma_{\text{Bool}}$: $+(v_0, v_1)$, $+(1, *(v_7, 0))$. They are usually written as $v_0 + v_1$ and $1 + (v_7 * 0)$.

Since terms are syntactic objects, they do not have a meaning. A term can be evaluated in a state, if elements of the superuniverse are assigned to the variables of the term.

**Definition 3.7 (Variable assignment).** Let $\mathfrak{A}$ be a state. A **variable assignment** for $\mathfrak{A}$ is a function $\zeta$ which assigns to each variable $v_i$ an element $\zeta(v_i) \in |\mathfrak{A}|$.

We write $\zeta\{x \mapsto a\}$ for the variable assignment which coincides with $\zeta$ except that it assigns the element $a$ to the variable $x$. So we have:

$$\zeta\{x \mapsto a\}(v_i) = \begin{cases} a, & \text{if } v_i = x \\ \zeta(v_i), & \text{otherwise.} \end{cases}$$

Given a variable assignment, the semantics of a term can be defined as an interpretation with respect to a state and a variable assignment in the traditional denotational way.

**Definition 3.8 (Interpretation of terms).** Let $\mathfrak{A}$ be a state of $\Sigma$, $\zeta$ be a variable assignment for $\mathfrak{A}$ and $t$ be a term of $\Sigma$. By induction on the length of $t$, a value $\llbracket t \rrbracket^\zeta_\mathfrak{A} \in |\mathfrak{A}|$ (the **interpretation of term** $t$ in state $\mathfrak{A}$) is defined as follows:

1. $\llbracket v_i \rrbracket^\zeta_\mathfrak{A} := \zeta(v_i)$ (interpretation of variables);
2. $\llbracket c \rrbracket^\zeta_\mathfrak{A} := \mathfrak{A}(c)$ (interpretation of constants);
3. $\llbracket f(t_1, \ldots, t_n) \rrbracket^\zeta_\mathfrak{A} := \mathfrak{A}(f)(\llbracket t_1 \rrbracket^\zeta_\mathfrak{A}, \ldots, \llbracket t_n \rrbracket^\zeta_\mathfrak{A})$ (interpretation of functions).

**Example 3.9.** Consider the state $\mathfrak{A}$ for $\Sigma_{\text{Bool}}$ of Example 3.4. Let $\zeta$ be a variable assignment with $\zeta(v_0) = 0$, $\zeta(v_1) = 1$ and $\zeta(v_2) = 1$. Then we have: $\llbracket v_0 + v_1 \rrbracket^\zeta_\mathfrak{A} = 1$.

**Definition 3.10 (Formulae).** Let $\Sigma$ be a vocabulary. A **formula** of $\Sigma$ is a syntactic expression generated as follows:

1. If $s$ and $t$ are terms of $\Sigma$, then $s = t$ is a formula.
2. If $\varphi$ is a formula, then $\neg \varphi$ is a formula.
3. If $\varphi$ and $\psi$ are formulae, then $(\varphi \land \psi)$, $(\varphi \lor \psi)$ and $(\varphi \rightarrow \psi)$ are formulae.
4. If $\varphi$ is a formula and $x$ a variable, then $(\forall x \varphi)$ and $(\exists x \varphi)$ are formulae.

A formula where all variables are quantified is a **closed formula**.

The logical connectives and quantifiers have the standard meaning. The expression $s = t$ is called an **equation**. The expression $s \neq t$ is an abbreviation for the formula $\neg(s = t)$. In order to increase the legibility of formulae, parentheses are often omitted (following the traditional left-to-right priorities).

The semantics of a formula is defined in the traditional way, i.e., by an interpretation with respect to a state and a variable assignment. Formulae are either true or false in a state. The truth value of a formula in a state is computed recursively. The classical truth tables for the logical connectives and the classical interpretation of quantifiers are used. The equality sign is interpreted as identity.
3.2 An Introduction to Abstract State Machines (ASMs)

Definition 3.11. Let $\mathfrak{A}$ be a state of $\Sigma$, $\varphi$ be a formula of $\Sigma$ and $\zeta$ be a variable assignment in $\mathfrak{A}$. By induction on the length of $\varphi$, a truth value $\llbracket \varphi \rrbracket^\mathfrak{A}_\zeta \in \{true, false\}$ (the interpretation of formula $\varphi$ in state $\mathfrak{A}$) is defined as follows:

$$\llbracket s = t \rrbracket^\mathfrak{A}_\zeta := \begin{cases} true, & \text{if } \llbracket s \rrbracket^\mathfrak{A}_\zeta = \llbracket t \rrbracket^\mathfrak{A}_\zeta; \\ false, & \text{otherwise}. \end{cases}$$

$$\llbracket \neg \varphi \rrbracket^\mathfrak{A}_\zeta := \begin{cases} true, & \text{if } \llbracket \varphi \rrbracket^\mathfrak{A}_\zeta = false; \\ false, & \text{otherwise}. \end{cases}$$

$$\llbracket \varphi \land \psi \rrbracket^\mathfrak{A}_\zeta := \begin{cases} true, & \text{if } \llbracket \varphi \rrbracket^\mathfrak{A}_\zeta = true \text{ and } \llbracket \psi \rrbracket^\mathfrak{A}_\zeta = true; \\ false, & \text{otherwise}. \end{cases}$$

$$\llbracket \varphi \lor \psi \rrbracket^\mathfrak{A}_\zeta := \begin{cases} true, & \text{if } \llbracket \varphi \rrbracket^\mathfrak{A}_\zeta = true \text{ or } \llbracket \psi \rrbracket^\mathfrak{A}_\zeta = true; \\ false, & \text{otherwise}. \end{cases}$$

$$\llbracket \varphi \rightarrow \psi \rrbracket^\mathfrak{A}_\zeta := \begin{cases} true, & \text{if } \llbracket \varphi \rrbracket^\mathfrak{A}_\zeta = false \text{ or } \llbracket \psi \rrbracket^\mathfrak{A}_\zeta = true; \\ false, & \text{otherwise}. \end{cases}$$

$$\llbracket \forall x \varphi \rrbracket^\mathfrak{A}_\zeta := \begin{cases} true, & \text{if } \llbracket \varphi \rrbracket^\mathfrak{A}_{\zeta \rightarrow \alpha} = true \text{ for all } \alpha \in |\mathfrak{A}|; \\ false, & \text{otherwise}. \end{cases}$$

$$\llbracket \exists x \varphi \rrbracket^\mathfrak{A}_\zeta := \begin{cases} true, & \text{if } \llbracket \varphi \rrbracket^\mathfrak{A}_{\zeta \rightarrow \alpha} = true \text{ for some } \alpha \in |\mathfrak{A}|; \\ false, & \text{otherwise}. \end{cases}$$

We say that a state $\mathfrak{A}$ is a model of $\varphi$ if $\llbracket \varphi \rrbracket^\mathfrak{A}_\zeta = true$ for all variable assignments $\zeta$.

3.2.3 Transition rules, consistent updates, firing of updates

In mathematics, states like Boolean algebras are static. They do not change over time. In computer science, states are dynamic. They evolve by being updated during computations. Updating abstract states means to change the interpretation of (some of) the dynamic functions in the underlying signature.

- In case of monitored functions, the system cannot change the interpretation (only the environment).
- In case of controlled functions, the system is allowed to update the interpretation of the function (and not the environment).

The way ASMs update states is described by transitions rules of Table 3.1 which define the syntax of ASM programs.

Definition 3.12 (Transition rules). Let $\Sigma$ be a vocabulary. The (transition) rules $R, S$ of an ASM are syntactic expressions generated according to Table 3.1.

Note that transition rules in the upper part of Table 3.1 are elementary constructs while transition rules in the lower part of Table 3.1 are derived constructs (syntactic sugar).

The semantics of transition rules is given by sets of updates. Since due to parallelism (in the Block and the Forall rules), a transition rule may prescribe to update the same function at the same arguments several times, we require such updates to be consistent. The concept of consistent update sets is made more precise by the following definitions.

Definition 3.13 (Update). An update for $\mathfrak{A}$ is a triple $(f, (a_1, \ldots, a_n), b)$, where $f$ is an $n$-ary dynamic function name, and $a_1, \ldots, a_n$ and $b$ are elements of $|\mathfrak{A}|$. An update set $U$ is a set of updates.

We frequently abbreviate $(a_1, \ldots, a_n)$ as $\bar{a}$ in the sequel.
### Table 3.1. Transition rules of ASMs

<table>
<thead>
<tr>
<th>Rule name</th>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip</td>
<td>skip</td>
<td>Do nothing.</td>
</tr>
<tr>
<td>Update</td>
<td>( f(t_1, \ldots, t_n) := s )</td>
<td>Syntactic conditions: (i) ( f ) is an ( n )-ary, dynamic function name of ( \Sigma ), (ii) ( t_1, \ldots, t_n ) and ( s ) are terms of ( \Sigma ). Meaning: In the next state, the value of the function ( f ) at the arguments ( t_1, \ldots, t_n ) is updated to ( s ). It is allowed that ( f ) is a ( 0 )-ary function. In this case, the update has the form ( c := s ).</td>
</tr>
<tr>
<td>Block</td>
<td>( R \ S )</td>
<td>( R ) and ( S ) are executed in parallel.</td>
</tr>
<tr>
<td>Conditional</td>
<td>( \text{if } \varphi \text{ then } R \text{ else } S )</td>
<td>If ( \varphi ) is true, then execute ( R ), otherwise execute ( S ).</td>
</tr>
<tr>
<td>Let</td>
<td>( \text{let } x = \ell \text{ in } R )</td>
<td>Assign the value of ( \ell ) to ( x ) and execute ( R ).</td>
</tr>
<tr>
<td>Forall</td>
<td>( \text{forall } x \text{ with } \varphi \text{ do } R )</td>
<td>Execute ( R ) in parallel for each ( x ) satisfying ( \varphi ).</td>
</tr>
<tr>
<td>Sequence</td>
<td>( R_1 ; S ) or ( R \text{ seq } S )</td>
<td>Execute first ( R ) and then ( S ) as an “atomic” action.</td>
</tr>
<tr>
<td>Iteration</td>
<td>( \text{iterate}(R) )</td>
<td>Execute the sequential composition of ( R ) with itself as long as possible.</td>
</tr>
<tr>
<td>Exception</td>
<td>( \text{try } R \text{ catch } f(t_1, \ldots, t_n) \ S )</td>
<td>Either the execution of ( R ) is consistent, or ( R ) is inconsistent but the location determined by ( f(t_1, \ldots, t_n) ) is not updated inconsistently. Otherwise it is defined by ( S ) (error code).</td>
</tr>
<tr>
<td>Rule definition</td>
<td>( r(x_1, \ldots, x_n) = \text{body} ) |</td>
<td></td>
</tr>
<tr>
<td>Rule definition</td>
<td>( R(x_1, \ldots, x_n) = \text{body}[] / / \text{re.subst} ) | A rule definition for a rule name ( r ) of arity ( n ) is an expression ( r(x_1, \ldots, x_n) = \text{body} ), where ( \text{body} ) is a transition rule.</td>
<td></td>
</tr>
<tr>
<td>Rule call</td>
<td>( r(t_1, \ldots, t_n) ) |</td>
<td></td>
</tr>
<tr>
<td>Rule call</td>
<td>( l := r(t_1, \ldots, t_n)) | In a rule call ( r(t_1, \ldots, t_n) ) the variables ( x_i ) in the body ( \text{body} ) of the rule definition are replaced by the parameters ( t_i ).</td>
<td></td>
</tr>
<tr>
<td>Choose</td>
<td>( \text{choose}(x) ) with ( \varphi ) do ( R ) = |</td>
<td></td>
</tr>
<tr>
<td>Choose</td>
<td>( \text{let } x = f_\varphi(\ldots) \text{ in } R )</td>
<td>Select non-deterministically an element ( x ) satisfying condition ( \varphi ) and execute ( R ) (( f_\varphi ) is a monitored choice function updated by the environment).</td>
</tr>
<tr>
<td>Create</td>
<td>( \text{create}(x) ) do ( R ) = |</td>
<td></td>
</tr>
<tr>
<td>Create</td>
<td>( \text{let } x = f_{\text{new}}(\ldots) \text{ in } R )</td>
<td>Create a new element ( x ) of the superuniverse which does not belong to any of the subuniverses (( f_{\text{new}}(\ldots) ) is a monitored function possibly with parameters)</td>
</tr>
<tr>
<td>While</td>
<td>( \text{while}(\varphi) \ R = |</td>
<td></td>
</tr>
<tr>
<td>While</td>
<td>( \text{iterate} ) (if ( \varphi ) then ( R ))</td>
<td>Repeat the execution of ( R ) as long as condition ( \varphi ) holds.</td>
</tr>
<tr>
<td>Do-Until</td>
<td>( \text{do } R \text{ until } (\neg \varphi) = |</td>
<td></td>
</tr>
<tr>
<td>Do-Until</td>
<td>( R \text{ seq while}(\neg \varphi) \ R )</td>
<td>First execute body ( R ) once and then behave like in case of ( \text{while} ) with negated condition ( \neg \varphi ).</td>
</tr>
</tbody>
</table>

The meaning of the update is that the interpretation of the function \( f \) in \( \mathcal{E} \) has to be changed at the arguments \( a_1, \ldots, a_n \) to the value \( b \). The pair of the first two components of an update is called a location. An update specifies how the function table of a dynamic function has to be updated at the corresponding location.

In a given state, a transition rule of an ASM produces for each variable assignment an update set. Since the rule can contain recursive calls to other rules, it is possible that it has no semantics at all. It can happen also that an update set contains several updates for the same function name \( f \). In this case, the updates have to be consistent, otherwise the execution stops.

**Definition 3.14 (Consistent update set).** An update set \( U \) is called consistent, if it satisfies the following property:

\[ (f,\bar{a},b) \in U \text{ and } (f,\bar{a},c) \in U, \text{ then } b = c. \]

This means that a consistent update set contains for each function and each argument tuple at most one value. Otherwise, the update set is called inconsistent.

**Example 3.15 (Consistency of update sets).** Let \( \text{curr} \) be a unary function symbol. The update set \( U \) consisting of three updates \( \{(\text{curr}, s_1, \top), (\text{curr}, s_2, \bot), (\text{curr}, s_1, \bot)\} \) is inconsistent since two different values \( (\top, \bot) \) are assigned to the same location \( (\text{curr}, s_1) \).
The update set \{\langle curr, s1, \top \rangle, \langle curr, s2, \bot \rangle, \langle curr, s1, \top \rangle\} is consistent. Although it contains two updates for location \langle curr, s1 \rangle, the values of these updates are identically \top.

If an update set \( U \) is consistent, it can be fired in a given state \( \mathcal{A} \) resulting in a new state \( \mathcal{B} \) in which the interpretations of dynamic function names are changed according to \( U \). The interpretations of static function names are the same as in the old state. The interpretation of monitored functions is given by the environment and can therefore change in an arbitrary way.

**Definition 3.16 (Firing of updates).** The result of firing a consistent update set \( U \) in a state \( \mathcal{A} \) is a new state \( \mathcal{B} \) (denoted as \( \mathcal{B} = \text{fire}_\mathcal{A}(U) \)) with the same superuniverse as \( \mathcal{A} \) satisfying the following two conditions for the interpretations of function names \( f \) of \( \Sigma \):

1. If \( (f, \bar{a}, b) \in U \), then \( f^\mathcal{B}(\bar{a}) = b \)
2. If there is no \( b \) with \( (f, \bar{a}, b) \in U \) and \( f \) is not a monitored function, then \( f^\mathcal{B}(\bar{a}) = f^\mathcal{A}(\bar{a}) \).

Firing an inconsistent update set is not allowed, i.e., \( \text{fire}_\mathcal{A}(U) \) is not defined for inconsistent \( U \).

Since \( U \) is consistent, for static and controlled functions the state \( \mathcal{B} \) is determined in a unique way. Notice that only those locations can have a new value in state \( \mathcal{B} \) with respect to state \( \mathcal{A} \) for which there is an update in \( U \). (In this way ASMs avoid the so called frame problem.)

Definition 3.16 yields the following (partial) next state function \( \text{next}_R \) which describes one application of a rule \( R \) in a state with a given environment (assignment) function. We often write \( \text{next}(R) \) instead of \( \text{next}_R \).

**Definition 3.17 (Next state function).** Let \( \mathcal{A} \) be a state, and \( \zeta \) an environment (assignment) function. The (partial) next state function \( \text{next}_R \) is defined as follows. \( \text{next}_R(\mathcal{A}, \zeta) = \text{fire}_\mathcal{A}(\llbracket R \rrbracket_\mathcal{A}) \).

Here \( \llbracket R \rrbracket_\mathcal{A} \) denotes the semantics of rule \( R \) which is defined as follows.

**Definition 3.18 (Semantics of transition rules).** The semantics of a(n elementary) transition rule \( R \) of a given ASM in a state \( \mathcal{A} \) with respect to a variable assignment \( \zeta \) is defined if and only there exists an update set \( U \) such that \( \llbracket R \rrbracket_\mathcal{A} \triangleright U \) can be derived by the semantic rules of Table 3.2. In that case \( \llbracket R \rrbracket_\mathcal{A} \) is identified with \( U \).

In order to complete Def. 3.18, we discuss the precise handling of the sequential composition and iteration.

**Sequential composition of rules**

Two update sets \( U \) and \( V \) of two consecutive steps can be merged (denoted as \( U \oplus V \)) where an update in \( V \) overwrites an update in \( U \) if this is for the same location since through a destructive assignment \( s := t \) the previous value of \( s \) is lost. We merge an update set \( V \) with \( U \) (i.e., \( U \oplus V \)) only if \( U \) is consistent otherwise we stick to \( U \) as we want both \( \text{fire}_\mathcal{A}(U) \) and \( \text{fire}_\mathcal{A}(U \oplus V) \) to be undefined.

**Definition 3.19 (Merging of update sets).** The merging of two update sets \( U \) and \( V \) is defined as follows.

\[
U \oplus V = \begin{cases} 
\{(f, \bar{a}, b) \mid (f, \bar{a}, b) \in U \land \nexists c : (f, \bar{a}, c) \in V\} \cup V, & \text{if } U \text{ is consistent;} \\
U, & \text{otherwise.}
\end{cases}
\]

The following propositions (taken from [30]) capture some elementary properties of the sequence operator.

**Proposition 3.20 (Persistence of inconsistency).** If \( \llbracket R \rrbracket_\mathcal{A} \) is not consistent then \( \llbracket R \text{ seq } S \rrbracket_\mathcal{A} = \llbracket R \rrbracket_\mathcal{A} \).
The next proposition shows that the seq construct captures the intended classical meaning of sequential composition of machines, if we look at them as state transforming functions.

**Proposition 3.21 (Compositionality of seq).** \( \text{next}(R \text{ seq } S) = \text{next}(R) \circ \text{next}(S) \).

**Proposition 3.22.** The seq construct has a left and right neutral element and is associative, i.e., for rules \( P, Q, \) and \( R \) the following holds:

1. \( \llbracket \text{skip} \text{ seq } P \rrbracket^\# = \llbracket P \text{ skip} \rrbracket^\# = \llbracket P \rrbracket^\# \)

2. \( \llbracket P \text{ seq } (R \text{ seq } S) \rrbracket^\# = \llbracket (P \text{ seq } R) \text{ seq } S \rrbracket^\# \).

**Iteration**

Once the sequence operator is defined, one can apply it repeatedly to define the iteration of a rule. This provides a natural way to define an iteration construct for ASMs which encapsulates the computation of a finite but a priori not explicitly known number of iterated steps into an atomic action (one-step computation).

**Definition 3.23.** The intention of rule iteration is to execute the given rule as long as possible. We define

\[
R^n = \begin{cases} 
\text{skip} & n = 0 \\
R^{n-1} \text{ seq } R & n > 0 
\end{cases}
\]

By \( \mathfrak{A}_n \), we denote the state obtained by firing the update set of the rule \( R^n \) in state \( \mathfrak{A} \), if defined (i.e., \( \mathfrak{A}_n = \text{next}_{R^n}(\mathfrak{A}) \)).

There are two natural stop situations for iterated ASM rule application, namely, when the update set becomes empty (the case of successful termination), and when it becomes inconsistent (in the case of failure), given the persistence of inconsistency as formulated in Proposition 3.20. Both cases provide a fixpoint \( \lim_{n \to \infty} \llbracket R^n \rrbracket^\# \) for the sequence \( (\llbracket R^n \rrbracket^\#)_{n>0} \) which becomes constant if a number \( n \) is found where the update set \( R \), in the state obtained by firing \( R^{n-1} \), is empty or inconsistent.

**Proposition 3.24 (Fixpoint condition).** \( \forall m \geq n > 0 \) the following holds: if \( \llbracket R \rrbracket^\#_{n-1} \) is not consistent or if it is empty, then \( \llbracket R^n \rrbracket^\# = \llbracket R^n \rrbracket^\# \).
The sequence \((\lfloor \mathit{P}^n \rfloor^3)_{n>0}\) eventually becomes constant only upon termination or failure. Otherwise the computation diverges and the update set for the iteration is undefined.

3.2.4 Abstract state machines

Finally, we can define the notion of abstract state machines, and the run of an ASM.

**Definition 3.25 (Abstract state machine, ASM).** An abstract state machine \(M\) consists of a vocabulary \(\Sigma\), an initial state \(A\) for \(\Sigma\), a rule definition for each rule name, and a distinguished rule name of arity zero called the main rule name of the machine.

An ASM is executed by firing repeatedly consistent update sets yielding a run of the ASM. The execution starts from the initial state, and terminates in a state where no transition rules are applicable (successful termination), or a rule application becomes inconsistent (abnormal termination).

**Definition 3.26 (Run of an ASM).** Let \(M\) be an ASM with vocabulary \(\Sigma\), initial state \(A\), and main rule name \(r\). Let \(\zeta\) be a variable assignment. A run of \(M\) is a finite or infinite sequence \(\mathcal{B}_0, \mathcal{B}_1, \ldots\) of states for \(\Sigma\) such that the following conditions are satisfied:

1. Initial state: \(\mathcal{B}_0 = A\).
2. Finite run: If \([r]^{\mathcal{B}}_{\zeta}\) is not defined or inconsistent, then \(\mathcal{B}_n\) is the last state in the sequence.
3. Infinite run: Otherwise, \(\mathcal{B}_{n+1}\) is the result of firing \([r]^{\mathcal{B}}_{\zeta}\) in \(\mathcal{B}_n\) (i.e., \(\mathcal{B}_{n+1} = \text{next}(\mathcal{B}_n)\)).

**Example 3.27 (ASM rules of traffic lights).** To demonstrate the dynamic behavior of ASMs, let the vocabulary \(\Sigma_{\text{traffic}}\) of a (Hungarian) traffic light consist of three nullary dynamic functions \{red, yellow, green\}. Initially, in state \(A\), let us define the following evaluation \([\text{red}]^A = \top\), \([\text{yellow}]^A = \bot\), and \([\text{green}]^A = \bot\).

1. The rule if \(\text{red}\) then \(\text{yellow} := \top\) should switch on the yellow light in addition to the red.
2. The rule if \(\text{red} \land \text{yellow}\) then \(\text{red} := \bot\); \(\text{yellow} := \bot\); \(\text{green} := \top\) should switch on green and switch off the other two lights.
3. The rule if \(\text{green}\) then \(\text{green} := \bot\); \(\text{yellow} := \top\) should switch off green and switch on yellow.
4. The rule if \(\neg \text{red} \land \text{yellow}\) then \(\text{red} := \top\); \(\text{yellow} := \bot\) switch on red and switch off the yellow light.

When applying Rule 1 in state \(A\), the update set \(U\) is \(\{(\text{yellow}, \top)\}\) the next state \(\mathcal{B}\) is \([\text{red}]^\mathcal{B} = \top\), \([\text{yellow}]^\mathcal{B} = \top\), and \([\text{green}]^\mathcal{B} = \bot\).

Now if we apply Rule 2 in state \(\mathcal{B}\) the update set \(U\) is \(\{(\text{red}, \bot), (\text{yellow}, \bot), (\text{green}, \top)\}\) the next state \(\mathcal{C}\) is \([\text{red}]^\mathcal{C} = \bot\), \([\text{yellow}]^\mathcal{C} = \bot\), and \([\text{green}]^\mathcal{C} = \top\).

3.2.5 An abstraction/refinement technique for abstract state machines

It is nowadays a common place that software design has to be hierarchical and has to be based on techniques for crossing the abstraction levels encountered on the long way from the understanding of the problem to the validation of its final solution. There are numerous proposals for defining and relating these levels as support for the separation of different software development concerns: separating program design from its implementation, from its verification, from its domain dependence, from hardware/software-partitioning, system design from software design from coding and similarly for testing, functionality from communication, etc. Numerous vertical structuring principles have been defined in terms of abstraction and refinement.

Various practical refinement notions have been developed by Börger et al. for abstract state machines (we basically follow [33] for their overview) through real-life case studies, in an attempt to
close, in a controllable manner, the gap between the design levels involved. They can all be put into the form of the well-known commuting diagram of Fig. 3.2, for a given machine $A$ which is refined to a machine $B$, where a usually partial abstraction function $\mathcal{F}$ serves as proof map, mapping certain refined states $\mathcal{B}$ of $B$ to abstract states $\mathcal{F}(\mathcal{B})$ of $A$, and certain sequences $R$ of $B$-rules to sequences $\mathcal{F}(R)$ of abstract $A$-rules (in cases where the proof map is used as refinement function, $\mathcal{F}$ goes from $A$ to $B$).

$$
\begin{array}{c}
A \xrightarrow{\mathcal{F}(R)} A' \\
\mathcal{F} \\
\downarrow \\
B \xrightarrow{R} B'
\end{array}
$$

Fig. 3.2. ASM refinement scheme

In order to establish the desired operational equivalence of the two machines, before proving the commutativity of the diagram, one can (and first of all has to) define the appropriate notions of correctness (i.e., each concrete computation implements an abstract computation) and/or completeness (i.e., each abstract computation is implemented by a concrete computation) between refined runs ($\mathcal{B}; S$) and abstract runs ($\mathcal{A}; R$). This definition is in terms of the locations (the “observables”) one wants to compare in the related states of the two machines. The observables could be, for example, the operations the user sees in the abstract machine, which are implemented through the refinement step.

We will frequently use this refinement scheme throughout the current thesis for proving correctness (and/or completeness) of certain constructions.

3.3 A Meta-Level Algebraic Representation of VPM Models

Based upon the previous definitions of abstract state machines we first define a meta-level algebraic representation of VPM models. Here the term “meta-level” refers to the fact that we use a fixed set of predefined function symbols uniformly representing classes and instances.

3.3.1 Vocabulary and state of VPM

Definition 3.28 (Meta-level Vocabulary of VPM). The (meta-level) vocabulary $\Sigma_{vpm}$ of our VPM framework is assumed to contain the following characteristic functions. (Arities of non-nullary functions are denoted by dashes in a Prolog style).

- $\text{undef, true, false}$ for the undefined symbol and elementary logic constants;
- $\text{entity}/1, \text{connection}/1, \text{mapping}/1$: unary functions for basic modelling concepts;
- $\text{from}/2, \text{to}/2, \text{supertype}/2, \text{instanceOf}/2, \text{componentOf}/2$: for relations/functions between basic modelling concepts.

Definition 3.29 (State of VPM). The superuniverse $\mathcal{A}$ of a state $\mathcal{A}$ of the VPM framework (i.e., of vocabulary $\Sigma_{vpm}$) contains the identifiers $\text{idp}$ of all potential model elements $P$. These identifiers (elements of $\mathcal{A}$) are either currently in use (i.e., $\exists P : \text{P.id = idp}$), or they are in the reserve.

The interpretation of function symbols of $\Sigma_{vpm}$ is as follows.
3.3 A Meta-Level Algebraic Representation of VPM Models

\[\text{entity}^3(id_E) := \begin{cases} \text{true}, & \text{if } \exists E : E.id = id_E; \\ \text{false}, & \text{otherwise}. \end{cases}\]

\[\text{connection}^3(id_R) := \begin{cases} \text{true}, & \text{if } \exists R(A,B) : R.id = id_R; \\ \text{false}, & \text{otherwise}. \end{cases}\]

\[\text{mapping}^3(id_F) := \begin{cases} \text{true}, & \text{if } \exists F(A,B) : F.id = id_F; \\ \text{false}, & \text{otherwise}. \end{cases}\]

\[\text{from}^3(id_X) := \begin{cases} \text{id}_A, & \text{if } \exists R(A,B) : R.id = \text{id}_X \land A.id = \text{id}_A \text{ or} \\ \exists F(A,B) : F.id = \text{id}_X \land A.id = \text{id}_A; \\ \text{undef}, & \text{otherwise}. \end{cases}\]

\[\text{to}^3(id_X) := \begin{cases} \text{id}_B, & \text{if } \exists R(A,B) : R.id = \text{id}_X \land B.id = \text{id}_B \text{ or} \\ \exists F(A,B) : F.id = \text{id}_X \land B.id = \text{id}_B; \\ \text{undef}, & \text{otherwise}. \end{cases}\]

\[\text{supertype}^3(id_X, id_Y) := \begin{cases} \text{true}, & \text{if } \exists X_{\text{sub}}, Y_{\text{super}} : X_{\text{sub}} \rightarrow Y_{\text{super}} \land X_{\text{sub}}.id = \text{id}_X \land \\ Y_{\text{super}}.id = \text{id}_Y; \\ \text{false}, & \text{otherwise}. \end{cases}\]

\[\text{instanceOf}^3(id_X, id_Y) := \begin{cases} \text{true}, & \text{if } \exists X_{\text{inst}}, Y_{\text{type}} : X_{\text{inst}} \mapsto Y_{\text{type}} \land X_{\text{inst}}.id = \text{id}_X \land \\ Y_{\text{type}}.id = \text{id}_Y; \\ \text{false}, & \text{otherwise}. \end{cases}\]

\[\text{componentOf}^3(id_X, id_E) := \begin{cases} \text{true}, & \text{if } \exists E, X, i : E^{(i)} = X \land E.id = \text{id}_E \land X.id = \text{id}_X; \\ \text{false}, & \text{otherwise}. \end{cases}\]

For a notational shorthand, the \(f(X) = \text{true}\) equations of characteristic functions will frequently be abbreviated to \(f(X)\) while \(\neg f(X)\) stands for \(f(X) = \text{false}\).

**Example 3.30.** The algebraic representation of the Graph package in Fig. 2.10 is listed in Table 3.3 (undefined and false locations are not listed).

<table>
<thead>
<tr>
<th>entity(id_graph) = true</th>
<th>entity(id_node) = true</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity(id_edge) = true</td>
<td>connection(id_from) = true</td>
</tr>
<tr>
<td>connection(id_from) = true</td>
<td>to(id_from) = id_node</td>
</tr>
<tr>
<td>from(id_from) = id_edge</td>
<td>to(id_to) = id_node</td>
</tr>
<tr>
<td>componentOf(id_node, id_graph) = true</td>
<td>componentOf(id_edge, id_graph) = true</td>
</tr>
<tr>
<td>componentOf(id_from, id_graph) = true</td>
<td>componentOf(id_to, id_graph) = true</td>
</tr>
</tbody>
</table>

**Table 3.3.** Algebraic (ASM) representation of graphs

For a notational shorthand, we will also use \(\text{entity}(\text{Graph})\) instead of \(\text{entity}(\text{id_graph})\) in the thesis even though it might introduce name clashes between function symbols and elements of the superuniverse (as in case of from and to).

### 3.3.2 Well-formedness constraints for the algebraic representation of VPM

Note that in general, ASMs do not impose any constraints on a state. However, the algebraic representation of a well-formed VPM model has to fulfill several requirements introduced in the following as invariants.
A VPM model is a graph

First, we discuss that the VPM model space forms a graph structure.

**Definition 3.31 (Graph).** By a graph we mean a directed unlabeled graph $G = (G_V, G_E, src^G, tar^G)$ with a set of nodes (vertices) $G_V$, a set of edges $G_E$, and functions $src^G : G_E \rightarrow G_V$ and $tar^G : G_E \rightarrow G_V$ associating to each edge its source and target nodes.

**Invariant 3.32 (Global graphs in VPM).** The VPM model is a graph where entities are nodes, and connections and mappings are well-formed edges. Formally, a VPM model is a graph $G_{vpm}$ with $G_V = \{ X \mid \text{entity}(X) \},$ and $G_E = \{ X \mid \text{connection}(X) \lor \text{mapping}(X) \}$ where

- A **connection** is a well-formed edge: $\forall R : \text{connection}(R) \rightarrow \exists A, B : \text{entity}(A) \land \text{from}(R) = A \land \text{entity}(B) \land \text{to}(R) = B.$ This formula will be referred to as $\varphi_{3.32}^{\text{conn}}.$

- A **mapping** is a well-formed edge: $\forall F : \text{mapping}(F) \rightarrow \exists A, B : \text{entity}(A) \land \text{from}(F) = A \land \text{entity}(B) \land \text{to}(F) = B.$ This formula will be referred to as $\varphi_{3.32}^{\text{map}}.$

As an abbreviation, we use $\varphi_{3.33}^{\text{dlo}} = \varphi_{3.32}^{\text{conn}} \land \varphi_{3.32}^{\text{map}}.$

The previous invariant introduces a global notion of graphs for the entire VPM model space, where entities are nodes, and connections/mappings are edges. On the other hand, we may introduce a local approach prescribing that each entity itself is a well-formed graph as well (thus obtaining a special kind of hierarchical graphs for the entire model space).

**Invariant 3.33 (Local graphs in VPM).** When interpreting graphs locally, a VPM entity itself is a graph where entities are nodes, and connections and mappings are edges all of them contained by the container entity itself. Formally, a VPM entity $Y$ is a graph $G_Y$ with $G_V = \{ X \mid \text{entity}(X) \land \text{componentOf}(X, Y) \},$ and $G_E = \{ X \mid \text{connection}(X) \lor \text{mapping}(X) \} \land \text{componentOf}(X, Y)$

- **Connection is an edge** in $Y$: $\forall R, E : \text{connection}(R) \land \text{componentOf}(R, E) \rightarrow \exists A, B : \text{entity}(A) \land \text{from}(R) = A \land \text{componentOf}(A, E) \land \text{entity}(B) \land \text{to}(R) = B \land \text{componentOf}(B, E).$ This formula will be referred to as $\varphi_{3.33}^{\text{conn}}.$

- **Mapping is an edge** in $Y$: $\forall F, E : \text{mapping}(F) \land \text{componentOf}(F, E) \rightarrow \exists A, B : \text{entity}(A) \land \text{from}(F) = A \land \text{componentOf}(A, E) \land \text{entity}(B) \land \text{to}(F) = B \land \text{componentOf}(B, E).$ This formula will be referred to as $\varphi_{3.33}^{\text{map}}.$

As an abbreviation, we use $\varphi_{3.33}^{\text{delo}} = \varphi_{3.33}^{\text{conn}} \land \varphi_{3.33}^{\text{map}}.$

Note that Invariant 3.33 is stronger than Invariant 3.32 in the sense that whenever a VPM model space forms a local graph then it forms a global graph as well.

**Refinement of VPM elements**

Now we formalize the axioms of the refinement calculus of Def. 2.12 in terms of the algebraic representation. This invariant states that the truth values of the supertype and instanceOf predicates (Boolean functions) should be synchronous with the refinement axioms.

**Invariant 3.34 (Refinement).** The axioms of the refinement calculus are formalized by the following invariants:

1. $\forall X, Y : \text{supertype}(X, Y) \rightarrow \text{isYSupertypeOf}(X, Y)$ (denoted as $\varphi_{3.34}^{\text{super}}$) where

   \[
   \text{isYSupertypeOf}(X, Y) \overset{\text{def}}{=} \text{entity}(X) \land \text{entity}(Y) \land \forall B_Y : \text{componentOf}(B_Y, Y) \rightarrow \exists A_X : \text{componentOf}(A_X, X) \land \text{supertype}(A_X, B_Y) \]

   - **Entity inheritance**: $(\text{entity}(X) \land \text{entity}(Y) \land \forall B_Y : \text{componentOf}(B_Y, Y) \rightarrow \exists A_X : (\text{componentOf}(A_X, X) \land \text{supertype}(A_X, B_Y)) \lor$
3.3 A Meta-Level Algebraic Representation of VPM Models

- **Connection inheritance:** $(\text{connection}(X) \land \text{connection}(Y) \land \exists F_X, \exists F_Y, \exists T_X, \exists T_Y : \text{from}(X) = F_X \land \text{entity}(F_X) \land \text{from}(Y) = F_Y \land \text{entity}(F_Y) \land \text{to}(X) = T_X \land \text{entity}(T_X) \land \text{to}(Y) = T_Y \land \text{entity}(T_Y) \land \text{supertype}(F_X, F_Y) \land \text{supertype}(T_X, T_Y)) \lor$

- **Mapping inheritance:** $(\text{mapping}(X) \land \text{mapping}(Y) \land \exists F_X, \exists F_Y, \exists T_X, \exists T_Y : \text{from}(X) = F_X \land \text{entity}(F_X) \land \text{from}(Y) = F_Y \land \text{entity}(F_Y) \land \text{to}(X) = T_X \land \text{entity}(T_X) \land \text{to}(Y) = T_Y \land \text{entity}(T_Y) \land \text{supertype}(F_X, F_Y) \land \text{supertype}(T_X, T_Y)) \lor$

- **Mapping is connection:** $(\text{mapping}(X) \land \text{connection}(Y) \land \exists F_X, \exists F_Y, \exists T_X, \exists T_Y : \text{from}(X) = F_X \land \text{entity}(F_X) \land \text{from}(Y) = F_Y \land \text{entity}(F_Y) \land \text{to}(X) = T_X \land \text{entity}(T_X) \land \text{to}(Y) = T_Y \land \text{entity}(T_Y) \land \text{supertype}(F_X, F_Y) \land \text{supertype}(T_X, T_Y))$

2. $\forall X, Y : \text{instanceOf}(X, Y) \rightarrow \text{isXInstanceOfY}(X, Y)$ (denoted as $\varphi^{\text{inst}}_{3,31}$) where

\[ \text{isXInstanceOfY}(X, Y) \equiv \]

- **Entity instantiation:** $(\text{entity}(X) \land \text{entity}(Y) \land \forall A_X : \text{componentOf}(A_X, X) 
\exists B_Y : (\text{componentOf}(B_Y, Y) \land \text{instanceOf}(A_X, B_Y)) \lor$

- **Connection instantiation:** $(\text{connection}(X) \land \text{connection}(Y) \land \exists F_X, \exists F_Y, \exists T_X, \exists T_Y : \text{from}(X) = F_X \land \text{entity}(F_X) \land \text{from}(Y) = F_Y \land \text{entity}(F_Y) \land \text{to}(X) = T_X \land \text{entity}(T_X) \land \text{to}(Y) = T_Y \land \text{entity}(T_Y) \land \text{supertype}(F_X, F_Y) \land \text{instanceOf}(T_X, T_Y)) \lor$

- **Mapping instantiation:** $(\text{mapping}(X) \land \text{mapping}(Y) \land \exists F_X, \exists F_Y, \exists T_X, \exists T_Y : \text{from}(X) = F_X \land \text{entity}(F_X) \land \text{from}(Y) = F_Y \land \text{entity}(F_Y) \land \text{to}(X) = T_X \land \text{entity}(T_X) \land \text{to}(Y) = T_Y \land \text{entity}(T_Y) \land \text{supertype}(F_X, F_Y) \land \text{instanceOf}(T_X, T_Y)) \lor$

- **Mapping is connection:** $(\text{mapping}(X) \land \text{connection}(Y) \land \exists F_X, \exists F_Y, \exists T_X, \exists T_Y : \text{from}(X) = F_X \land \text{entity}(F_X) \land \text{from}(Y) = F_Y \land \text{entity}(F_Y) \land \text{to}(X) = T_X \land \text{entity}(T_X) \land \text{to}(Y) = T_Y \land \text{entity}(T_Y) \land \text{instanceOf}(F_X, F_Y) \land \text{instanceOf}(T_X, T_Y))$

As an abbreviation, we use $\varphi^{\text{refl}}_{3,31} = \varphi^{\text{super}}_{3,31} \land \varphi^{\text{inst}}_{3,31}$

**Refinement relation is partial order**

The next invariants express that refinement relations must not contain circular dependencies. Intuitively, it means that, for instance, cycles are not allowed in the inheritance hierarchy of classes as in case of traditional object-oriented programming languages.

**Invariant 3.35.** The boolean function \text{supertype} imposes a partial order on elements, i.e., it is reflexive, transitive and anti-symmetric. Formally, $\varphi^{\text{po}}_{3,35} = \varphi^{\text{refl}}_{3,35} \land \varphi^{\text{trans}}_{3,35} \land \varphi^{\text{anti}}_{3,35}$

- **reflexive:** $\forall X : (\text{entity}(X) \lor \text{connection}(X) \lor \text{mapping}(X)) \iff \text{supertype}(X, X)$ (referred to as $\varphi^{\text{po}}_{3,35}$).
- **transitive:** $\forall X, Y, Z : \text{supertype}(X, Y) \land \text{supertype}(Y, Z) \rightarrow \text{supertype}(X, Z)$ (referred to as $\varphi^{\text{trans}}_{3,35}$).  
- **anti-symmetric:** $\forall X, Y : \text{supertype}(X, Y) \land \text{supertype}(Y, X) \rightarrow X = Y$ (referred to as $\varphi^{\text{anti}}_{3,35}$).

**Invariant 3.36.** The boolean function \text{instanceOf} imposes a partial order on elements, i.e., it is reflexive, transitive and anti-symmetric. Formally, $\varphi^{\text{po}}_{3,36} = \varphi^{\text{refl}}_{3,36} \land \varphi^{\text{trans}}_{3,36} \land \varphi^{\text{anti}}_{3,36}$

- **reflexive:** $\forall X : (\text{entity}(X) \lor \text{connection}(X) \lor \text{mapping}(X)) \iff \text{instanceOf}(X, X)$ (referred to as $\varphi^{\text{refl}}_{3,36}$).
- **transitive:** $\forall X, Y, Z : \text{instanceOf}(X, Y) \land \text{instanceOf}(Y, Z) \rightarrow \text{instanceOf}(X, Z)$ (referred to as $\varphi^{\text{trans}}_{3,36}$).
- **anti-symmetric:** $\forall X, Y : \text{instanceOf}(X, Y) \land \text{instanceOf}(Y, X) \rightarrow X = Y$ (referred to as $\varphi^{\text{anti}}_{3,36}$).

The component-of relation is transitive and reflexive

As for the \text{componentOf} relation, we do not prohibit circular dependencies — as done in case of inheritance and instantiation relations — in order to allow the declaration of recursive structures (like trees, lists, etc.). On the other hand, only entities are allowed to contain other model elements therefore the containment relation is not interpreted for connections and mappings.
Invariant 3.37. The componentOf relation is transitive for entities and reflexive for all model elements.

- reflexive: $\forall X : \text{componentOf}(X, X) \iff \text{entity}(X)$ (referred to as $\psi^{refl}_{3.37}$).
- transitive: $\forall X, Y, Z : \text{componentOf}(X, Y) \land \text{componentOf}(Y, Z) \rightarrow \text{componentOf}(X, Z)$ (referred to as $\psi^{trans}_{3.37}$).

As an abbreviation, we use $\psi^{comp}_{3.37} = \psi^{refl}_{3.37} \land \psi^{trans}_{3.37}$.

Consistent VPM state

Finally, we define when a VPM state is called consistent.

Invariant 3.38 (Consistent VPM state). A VPM state $\mathfrak{A}$ is called globally consistent (abbreviated as $\phi^{ge}_{3.38}$), if $\phi^{ge}_{3.38} = \phi^{loc}_{3.33} \land \phi^{ref}_{3.34} \land \phi^{po}_{3.35} \land \phi^{po}_{3.36} \land \phi^{comp}_{3.37}$.

A VPM state $\mathfrak{A}$ is called locally consistent (denoted as $\phi^{loc}_{3.33}$), if $\phi^{loc}_{3.33} = \phi^{ref}_{3.34} \land \phi^{po}_{3.35} \land \phi^{po}_{3.36} \land \phi^{comp}_{3.37}$.

When local or global consistency is irrelevant (i.e., its evaluation does not depend on the local or global interpretation) we use the following notation: $\phi^{cons}_{3.38}$.

Finally, in many parts of the current thesis, we also prescribe that a VPM model space should contain a top-most model element both in the inheritance and containment hierarchy. In other terms:

Invariant 3.39. Each model element has a supertype and it is contained by another model element.

- Inheritance: $\forall X : X \neq \text{id}_{univ} \rightarrow \exists Y : X \neq Y \land \text{supertype(X,Y)}$ (referred to as $\phi^{inh}_{3.39}$).
- Containment: $\forall X : X \neq \text{id}_{univ} \rightarrow \exists Y : X \neq Y \land \text{componentOf}(X,Y)$ (referred to as $\phi^{cont}_{3.39}$).

3.4 Elementary Manipulations of VPM Models

Now we define a set of elementary operations that consistently manipulate VPM models, i.e., they preserve the validity of the above invariants.

3.4.1 Manipulating VPM models: an overview

The elementary operations for manipulating VPM models can be grouped into the following main categories:

- Adding new model elements. We need to provide a means to create new entities, connections, mappings, or extend supertype, instance-of and component-of relations.
- Deleting existing elements. Additionally, model elements should be removed consistently (i.e., to obtain a consistent VPM model space). In case of removing entities, we define two sets of delete operations:
  - Soft delete. A soft delete operation is allowed to remove only the entity itself but none of the related connections or mappings. If such “dangling edges” (i.e., connections or mappings) would exist as a result of executing the operation then the operation is not executed at all.
  - Forced delete. Unlike soft delete, the forced delete (or hard delete) operation implicitly removes all dangling connections and mappings related to the entity to be removed.

As a VPM model element may be contained by multiple entities, from a practical point of view, it is worth introducing two distinct sets of operations depending whether an operation (especially a delete operation) should be executed globally or locally.

- Global manipulations. A global “delete” operation removes an element from the entire VPM model space, i.e., first from all container entities and then destroys the element itself. A global “add” operation only creates the model element but does not add it to any models.
• **Local manipulations.** A local "delete" operation removes an element from the content of some entities, and only destroys it if it is not contained (used) by any elements. A local "add" operation only creates the model element if does not exist (i.e., it is not part of any models).

As a summary, a global operation applied on a model element performs at least the manipulations that are prescribed by a local operation on the same element. In the sequel, one can easily notice the distinction between local and global manipulations as both local and global versions of these operations will be defined.

### 3.4.2 Auxiliary rules for manipulating VPM models

Now, prior to discussing the elementary manipulations themselves, we introduce some auxiliary functions for testing and properly handling (i) the refinement and containment hierarchy, and (ii) different design decisions on removing entities (i.e., soft or forced delete).

**Testing the isolation of entities**

Algorithm 1 checks whether an entity X is isolated (in the VPM model space, or in a given container entity Y). Isolation (by definition) means that no connections or mappings are leading from or leading to a given entity. Note that in the local version (isIsolatedIn Y), connections and mappings may lead from or to a given locally isolated entity provided that these elements are not part of the container entity Y.

\[
\text{Algorithm 1 Testing the isolation of entities}
\]

\[
\text{dependent function isIsolated}(X) : \mathbb{B} =
\]

1. entity(X) \land (\forall Y : (connection(Y) \lor mapping(Y)) \rightarrow (from(Y) \neq X \land to(Y) \neq X))

\[
\text{dependent function isIsolatedInE}(X, E) : \mathbb{B} =
\]

2. entity(X) \land entity(E) \land componentOf(X, E) \land (\forall Y : ((connection(Y) \lor mapping(Y)) \land componentOf(Y, E)) \rightarrow (from(Y) \neq X \land to(Y) \neq X))

While the isIsolated test will serve as a primary basis for soft delete operations, the upcoming rules in Algorithm 2 and 3 provide means to properly handle forced delete operation when all related connections and mappings should be removed together with the entity itself.

**Deleting the refinement hierarchy for a model element**

Rule deleteFromHierarchy (in Alg. 2) simply falsifies all refinement and containment locations leading from or to a model element P (which is generally applicable to entities, connections and mappings as well).

Now we formalize the correctness of applying rule deleteFromHierarchy.

**Proposition 3.40.** Applying rule deleteFromHierarchy (referred to as dfh) in a well-formed VPM state respects Invariants 3.35–3.37. Formally, \(\forall A, B : B = nextdfh(A) \land \left[\varphi_{3.36}^{\text{com}}\right]_A \rightarrow \left[\varphi_{3.35}^{\text{com}}\right]_B \land \left[\varphi_{3.30}^{\text{com}}\right]_B \land \left[\varphi_{3.37}^{\text{com}}\right]_B\).

**Corollary 3.41.** Applying rule deleteFromHierarchy (referred to as dfh) with parameter P removes all supertype, instance-of and component-of relations leading from or to element P. Formally, \(\forall A, B : B = nextdfh(A) \rightarrow \forall X : \text{supertype}(X, P) \lor \text{supertype}(P, X) \lor \text{instanceOf}(X, P) \lor \text{instanceOf}(P, X) \lor \text{componentOf}(X, P) \lor \text{componentOf}(P, X)\)
Algorithm 2 Deleting the refinement and containment hierarchy for a model element

\[
\text{deleteFromHierarchy}(P) = \\
1: \text{forall } Q \text{ with } P \neq Q \land \text{supertype}(P, Q) \text{ do} \\
2: \text{supertype}(P, Q) := \text{false} \\
3: \text{end for} \\
4: \text{forall } Q \text{ with } P \neq Q \land \text{supertype}(Q, P) \text{ do} \\
5: \text{supertype}(Q, P) := \text{false} \\
6: \text{end for} \\
7: \text{forall } Q \text{ with } P \neq Q \land \text{instanceOf}(P, Q) \text{ do} \\
8: \text{instanceOf}(P, Q) := \text{false} \\
9: \text{end for} \\
10: \text{forall } Q \text{ with } P \neq Q \land \text{instanceOf}(Q, P) \text{ do} \\
11: \text{instanceOf}(Q, P) := \text{false} \\
12: \text{end for} \\
13: \text{forall } Q \text{ with } P \neq Q \land \text{componentOf}(P, Q) \text{ do} \\
14: \text{componentOf}(P, Q) := \text{false} \\
15: \text{end for} \\
16: \text{forall } Q \text{ with } P \neq Q \land \text{componentOf}(Q, P) \text{ do} \\
17: \text{componentOf}(Q, P) := \text{false} \\
18: \text{end for} \\
\]

Deleting dangling edges

Rule \text{deleteDanglingEdgesOf}X (Alg. 3) is responsible for removing all connections and mappings that are related to an entity \(X\). Finally, the local version of the same rule (\text{deleteDanglingEdgesOf}YIn\ Y) only removes dangling elements from a given container entity \(Y\). Note that only related connections and mappings are considered to be dangling, but not containment relations.

Algorithm 3 Deleting dangling edges

\[
\text{deleteDanglingEdgesOf}X(X) = \\
1: \text{forall } Q \text{ with } \text{connection}(Q) \land (\text{from}(Q) = X \lor \text{to}(Q) = X) \text{ do} \\
2: \text{deleteConnection}(Q) \{ \text{To be defined later} \} \\
3: \text{end for} \\
4: \text{forall } Q \text{ with } \text{mapping}(Q) \land (\text{from}(Q) = X \lor \text{to}(Q) = X) \text{ do} \\
5: \text{deleteMapping}(Q) \{ \text{To be defined later} \} \\
6: \text{end for} \\
\]

\[
\text{deleteDanglingEdgesOf}YIn\ Y(X, Y) = \\
7: \text{forall } Q \text{ with } \text{connection}(Q) \land \text{entity}(Y) \land \text{componentOf}(Q, Y) \land (\text{from}(Q) = X \lor \text{to}(Q) = X) \text{ do} \\
8: \text{deleteConnection}XinY(Q, Y) \{ \text{To be defined later} \} \\
9: \text{end for} \\
10: \text{forall } Q \text{ with } \text{mapping}(Q) \land \text{entity}(Y) \land \text{componentOf}(Q, Y) \land (\text{from}(Q) = X \lor \text{to}(Q) = X) \text{ do} \\
11: \text{deleteMappingXinY}(Q, Y) \{ \text{To be defined later} \} \\
12: \text{end for} \\
\]

The following proposition establishes the correctness of rule \text{deleteDanglingEdgesOf}X.

Proposition 3.42. Supposing that \text{deleteConnection} (\text{deleteMapping}) soundly removes a connection (a mapping) from a VPM state (which we prove later in Prop. 3.65–3.66), the rule application of \text{deleteDanglingEdgesOf}X (referred to as \text{gde}(X)) on entity \(X\) in a well-formed VPM state respects Invariant 3.32. Formally, \(\forall \mathcal{A}, \mathcal{B} : \mathfrak{B} = \text{next}_{\text{gde}(X)}(\mathcal{A}) \land \|\mathcal{F}_{3.36\mathfrak{A}}\|_{\mathfrak{B}} \rightarrow \|\mathcal{F}_{3.32\mathfrak{B}}\|_{\mathfrak{B}} \land \|\text{isIsolated}(X)\|_{\mathfrak{B}}\).

Similarly, the rule application of \text{deleteDanglingEdgesOf}YIn\ Y on model elements \(X\) and \(Y\) (referred to as \text{ldde}(X,Y)) in a well-formed VPM state respects Invariant 3.32. Formally, \(\forall \mathcal{A}, \mathcal{B} : \mathfrak{B} = \text{next}_{\text{ldde}(X,Y)}(\mathcal{A}) \land \|\mathcal{F}_{3.36\mathfrak{A}}\|_{\mathfrak{B}} \rightarrow \|\mathcal{F}_{3.32\mathfrak{B}}\|_{\mathfrak{B}} \land \|\text{isIsolatedIn}(Y)\|_{\mathfrak{B}}\).
Checking consistency of refinement relations in VPM models

The role of Algorithm 4 is to synchronize the supertype and instanceOf relations in the algebraic representation of the VPM framework with the refinement axioms.

Algorithm 4 Resolving consistency of refinement relations

\begin{verbatim}
rule checkConsistency() =
1: if \exists X, Y : \neg isSupertypeOf(X, Y) \land supertype(X, Y) then
2: forall X, Y with \neg isSupertypeOf(X, Y) \land supertype(X, Y) do
3:   supertype(X, Y) := false
4: end for
5: checkConsistency()
6: else if \exists X, Y : \neg isInstanceOf(X, Y) \land instanceOf(X, Y) then
7: forall X, Y with \neg isInstanceOf(X, Y) \land instanceOf(X, Y) do
8:   instanceOf(X, Y) := false
9: end for
10: checkConsistency()
11: else
12:   skip
13: end if
\end{verbatim}

Algorithm 4 falsifies all locations in supertype and instanceOf relations (Lines 2 and 5) that violate the refinement axioms. More precisely, if a model element Y is not a supertype of X according to the axioms but supertype(X, Y) holds then this location should be falsified. Each time one or more new locations are falsified rule checkConsistency is called recursively (Line 3 and 6).

On the other hand, if a model element Y is a supertype of X according to the axioms but supertype(X, Y) does not hold then no locations are altered since refinement relations have to be introduced explicitly if required (by intention). This procedure is a consequence of the fact that the refinement invariants (Invariant 3.34) only prescribe necessary conditions for the supertype and instanceOf relations.

This time the correctness of rule checkConsistency is formalized by handling its termination and evaluating its effects.

**Proposition 3.43.** Algorithm 4 always terminates.

**Proposition 3.44.** The result of applying the ASM rule checkConsistency (abbreviated as cc) always fulfills Invariant 3.34 (regardless of the preceding state). Formally, \( \forall A, B : B = \text{next}_{cc}(A) \) \( \Rightarrow \varphi_{3.34}^{\text{super}} \land \varphi_{3.34}^{\text{inst}} \) .

### 3.4.3 Operations for model manipulation

**Modifying refinement relations**

Algorithm 4 implicitly removed all inconsistent refinement relations. Now, we define rules for explicitly modifying the refinements, i.e., consistently adding and removing supertype and instanceOf relations between VPM elements.

Since the handling of supertype and instanceOf relations are essentially the same, we only discuss in details the case of supertype in Algorithm 5 and omit the specification of rules for adding and removing instanceOf relations.

When adding a new inheritance relation between model elements X and Y (Y is intended to be supertype of X), we must first assure that the new supertype relation will not introduce circularities (Line 1 of Alg. 5) in the hierarchy. Then, a supertype relation should be introduced (in Line 2-4) from X to all supertypes Z of Y (including Y itself).
Algorithm 5 Adding and removing supertype relations

\begin{verbatim}
rule addYtoSupertypeOfX(X, Y) =
  1: if ¬supertype(Y, X) ∧ X ≠ Y then
  2:   forall Z with supertype(Y, Z) ∧ isYSupertypeOfX(X, Z) ∧ ¬supertype(X, Z) do
  3:     supertype(X, Z) := true
  4:   end for
  5: else {Precondition is false}
  6:   skip
  7: end if

rule delZfromSupertypeOfX(X, Z) =
  8: if supertype(X, Z) ∧ X ≠ Z then
  9:   forall Y with supertype(X, Y) ∧ supertype(Y, Z) do
 10:     supertype(X, Y) := false
 11:   end for
 12:   checkConsistency()
 13: else {Precondition is false}
 14:   skip
 15: end if
\end{verbatim}

When removing an existing inheritance relation (Line 8) between model elements X and Z (Z is parent of X), one should remove all supertype relations from X to all elements situated between X and Z in the inheritance hierarchy (Line 9–11). Since additional refinement relations may be violated in this case, we call the checkConsistency algorithm.

Now we establish the correctness of rule \textit{addYtoSupertypeOfX}.

**Proposition 3.45.** Applying rules \textit{addYtoSupertypeOfX} (referred to as \textit{add}) and \textit{delZfromSupertypeOfX} (referred to as \textit{del}) in a well-formed VPM state preserves Invariants 3.34 and 3.35. Formally, \(\forall A, B : (B = next_{add}(A)) \lor (B = next_{del}(A)) \land \llbracket \varphi_{3.34} \rrbracket_{C} \rightarrow \llbracket \varphi_{3.34}^{\supertype} \rrbracket_{C} \land \llbracket \varphi_{3.35}^{\supertype} \rrbracket_{C}\)

Note that the corresponding locations of the supertype relation are always removed but they are only set to true if the result is a consistent VPM model. Therefore the following corollaries are trivial consequences of Algorithm 5.

**Corollary 3.46.** Applying rule \textit{addYtoSupertypeOfX} (referred to as \textit{add}) with parameters \((X, Y)\) adds a corresponding supertype relation between \(X\) and \(Y\) provided that it does not introduce inconsistencies. Formally, \(\forall A, B : (B = next_{add}(A)) \land (X ≠ Y) \land \llbracket \neg \text{-supertype}(Y, X) \land \neg \text{-supertype}(X, Y) \land \neg \text{isYSupertypeOfX}(X, Y) \rrbracket_{C} \rightarrow \llbracket \text{supertype}(X, Y) \rrbracket_{C}^{\supertype}\).

**Corollary 3.47.** Applying rule \textit{delZfromSupertypeOfX} (referred to as \textit{del}) with parameters \((X, Z)\) removes the corresponding supertype relation between \(X\) and \(Z\) (if \(X\) and \(Z\) are not identical). Formally, \(\forall A, B : (B = next_{del}(A)) \land (X ≠ Z) \rightarrow \llbracket \neg \text{-supertype}(X, Z) \rrbracket_{C}^{\supertype}\).

**Handling component-of relations**

The handling of component-of relations as provided in Algorithm 6 is conceptually similar to manipulating supertype and instance-of relations.

If a model element \(X\) is intended to be added to an entity \(Y\) (where \(X ≠ Y\)) then \(X\) has to be added to all entities containing \(Y\); naturally including \(Y\) itself. On the other hand, if a model element \(X\) is intended to be removed from an entity \(Z\) (where \(X ≠ Z\) again) then \(X\) has to be deleted from all entities \(Y\) that are contained by \(Z\) including entity \(Z\) as well.

Note that the addition and deletion of component-of relations has impact on the refinement relations. For instance, adding a new component-of relation to a super-metamodel (i.e., adding a new class to a metamodel) may destroy currently existing inheritance relations to a sub-metamodel. Alternatively, removing an existing component-of relation from a metamodel may cause that an instance
Algorithm 6 Handling component containments

```
rule addXtoComponentY(X, Y) =
1: if X ≠ Y ∧ entity(Y) ∧ ¬componentOf(X, Y) then
2:   forall Z with entity(Z) ∧ componentOf(Y, Z) do
3:     componentOf(X, Z) := true
4:   end for
5:   checkConsistency()
6: else {Preconditions are false}
7:   skip
8: end if
rule delXfromComponentZ(X, Z) =
9: if X ≠ Z ∧ entity(Z) ∧ componentOf(X, Z) then
10:   forall Y with entity(Y) ∧ componentOf(Y, Z) ∧ componentOf(X, Y) do
11:     componentOf(X, Y) := false
12:   end for
13:   checkConsistency()
14: else {Preconditions are false}
15:   skip
16: end if
```

model is no longer a valid instance of this metamodel. Therefore, in each rule `addXtoComponentY` and `delXfromComponentZ`, we need an explicit call to `checkConsistency`.

**Proposition 3.48.** Applying rules `addXtoComponentY` (below referred to as `add`) and `delXfromComponentZ` (referred to as `del`) in a well-formed VPM state preserves Invariant 3.37. Formally, $∀A, B : (B = next_{add}(A) ∨ B = next_{del}(A)) ∧ \text{componentOf}(X, Y) ∧ \text{entity}(Y) \rightarrow \text{componentOf}(X, Y)_{\text{add}}$.

We can also establish this time as well the trivial consequences of Alg. 6 that it adds or removes the corresponding component-of relations between model elements passed as parameters.

**Corollary 3.49.** Applying rule `addXtoComponentY` (referred to as `add`) with parameters $(X, Y)$ adds a corresponding component-of relation between $X$ and $Y$ provided that $X \neq Y$ and $Y$ is an entity. Formally, $∀A, B : B = next_{add}(A) ∧ (X \neq Y) ∧ \text{componentOf}(X, Y) ∧ \text{entity}(Y) \rightarrow \text{componentOf}(X, Y)_{\text{add}}$.

**Corollary 3.50.** Applying rule `delXfromComponentZ` (referred to as `del`) with parameters $(X, Z)$ removes the corresponding component-of relation between $X$ and $Z$ provided that $X$ and $Z$ are not identical and $Z$ is an entity. Formally, $∀A, B : B = next_{del}(A) ∧ (X \neq Z) ∧ \text{componentOf}(X, Z) ∧ \text{entity}(Z) \rightarrow \text{componentOf}(X, Z)$.

Adding and deleting entities globally

Now we define the rules for globally manipulating (i.e., creating and destroying) entities in Algorithm 7–8. In this sense, when a non-existing entity is to be created by Alg. 7 (with an unused identifier given as parameter) it is first isolated from all existing elements (except from itself). Similarly, if an existing entity is to be removed by Alg. 8, either it should be isolated (in case of soft delete) or all dangling connections are removed (in case of forced delete).

**Proposition 3.51.** Applying rule `addEntity` (referred to as `ae`) in a well-formed VPM state preserves Invariant 3.38. Formally, $∀A, B : (B = next_{ae}(A) ∨ B = next_{del}(A)) ∧ \text{entity}(X) \rightarrow \text{componentOf}(X, X) ∧ \text{superType}(X, X) ∧ \text{instanceOf}(X, X)_{\text{ae}}$.

**Corollary 3.52.** Applying rule `addEntity` (referred to as `ae`) with parameter $X$ adds a corresponding entity $X$ provided that entity $X$ was non-existent before. Formally, $∀A, B : B = next_{ae}(A) ∧ \text{entity}(X) \rightarrow \text{componentOf}(X, X) ∧ \text{superType}(X, X) ∧ \text{instanceOf}(X, X)_{\text{ae}}$.
Algorithm 7 Adding entities to VPM
\begin{verbatim}
rule addEntity(X) =
1: if ¬entity(X) then
2: entity(X) := true;
3: componentOf(X, X) := true;
4: supertype(X, X) := true;
5: instanceOf(X, X) := true;
6: else {Precondition is false}
7: skip
8: end if
\end{verbatim}

Algorithm 8 Deleting entities from VPM
\begin{verbatim}
rule softDeleteEntity(X) =
1: if entity(X) ∧ isolated(X) then
2: deleteFromHierarchy(X);
3: componentOf(X, X) := false;
4: supertype(X, X) := false;
5: instanceOf(X, X) := false;
6: entity(X) := false;
7: checkConsistency();
8: else
9: skip
10: end if

rule forcedDeleteEntity(X) =
11: if entity(X) then
12: deleteDanglingEdgesOfX(X);
13: softDeleteEntity(X);
14: else
15: skip
16: end if
\end{verbatim}

When performing a soft delete on an entity \( X \) of a VPM model space, first this entity should be removed from the refinement and containment hierarchy (by calling deleteFromHierarchy) supposing that this entity is isolated. Then the entity location \( X \) is falsified and the refinement relations are refreshed (by calling checkConsistency) otherwise the removal of refinement and containment relations may result in an inconsistent model space. In case of a forced delete, the isolation of the entity to be removed is not checked; however, all dangling edges are explicitly deleted.

**Proposition 3.53.** Applying rule softDeleteEntity (referred to as sde) in a well-formed VPM state preserves Invariant 3.38. Formally, \( ∀A, B : B = next_{sde}(A) ∧ \llbracket \varphi_{3,38}^{\mathfrak{A}} \rrbracket_B \rightarrow \llbracket \varphi_{3,38}^{\mathfrak{B}} \rrbracket_B \).

**Corollary 3.54.** Applying rule softDeleteEntity (referred to as sde) with parameter \( X \) removes the corresponding entity \( X \) provided that entity \( X \) was existent and isolated before. Formally, \( ∀A, B : B = next_{sde}(A) ∧ \llbracket entity(X) ∧ isolated(X) \rrbracket_B \rightarrow \llbracket ¬entity(X) ∧ ¬componentOf(X, X) ∧ ¬supertype(X, X) ∧ ¬instanceOf(X, X) \rrbracket_B \).

**Proposition 3.55.** Applying rule forcedDeleteEntity (referred to as fde) in a well-formed VPM state preserves Invariant 3.38. Formally, \( ∀A, B : B = next_{fde}(A) ∧ \llbracket \varphi_{3,38}^{\mathfrak{A}} \rrbracket_B \rightarrow \llbracket \varphi_{3,38}^{\mathfrak{B}} \rrbracket_B \).

**Corollary 3.56.** Applying rule forcedDeleteEntity (referred to as fde) with parameter \( X \) removes the corresponding entity \( X \) provided that entity \( X \) was existent before. Formally, \( ∀A, B : B = next_{fde}(A) ∧ \llbracket entity(X) \rrbracket_B \rightarrow \llbracket ¬entity(X) ∧ ¬componentOf(X, X) ∧ ¬supertype(X, X) ∧ ¬instanceOf(X, X) \rrbracket_B \).
3.4 Elementary Manipulations of VPM Models

Adding and deleting entities locally

The local counterpart of manipulating entities are specified in Algorithm 9 and 10. As they are very similar to the global case, we only point out the conceptual differences and omit the proofs from Appendix B.1.

Algorithm 9 Adding entities to a model

```plaintext
rule addEntityXtoY(X, Y) =
1: if X ≠ Y ∧ entity(Y) ∧ ¬componentOf(X, Y) then {Y is an entity}
2: if ¬entity(X) then {Entity X does not exist}
3: addEntity(X)
4: end if
5: addXtoComponentY(X, Y);
6: checkConsistency();
7: else {Precondition is false}
8: skip
9: end if
```

When adding entity X to a container entity Y, we need to check whether entity X is already existent in the VPM model space (Line 2 in Alg. 9). If so, only componentOf locations are set to true; otherwise entity X itself is created as well.

Algorithm 10 Deleting entities from a model

```plaintext
rule softDeleteEntityXfromY(X, Y) =
1: if X ≠ Y ∧ entity(X) ∧ entity(Y) ∧ componentOf(X, Y) ∧ isXolatedInE(X, Y) then {Entity X is a component of Y}
2: delXfromComponentZ(X, Y)
3: if ∀Z : entity(Z) ∧ componentOf(X, Z) → X = Z then {Entity X is not contained by any elements but itself}
4: softDeleteEntity(X)
5: end if
6: checkConsistency();
7: else {Precondition is false}
8: skip
9: end if
```

Unlike the global case, when removing an entity X locally (from a container entity Y), it is essentially deleted only if X is not part of any models (compound entities) any more (see Lines 3–4 in Alg. 10).

The correctness of adding and deleting entities is captured by the following propositions.

**Proposition 3.57.** Applying rule addEntityXtoY (referred to as lce) in a well-formed VPM state preserves Invariant 3.38. Formally, ∀A, B : (A = next_lce(A) ∨ B = next_lce(B)) ∧ [[φ^lce]][A] → [[φ^lce]][B].

**Proposition 3.58.** Applying rule softDeleteEntityXfromY (referred to as lde) in a well-formed VPM state preserves Invariant 3.38. Formally, ∀A, B : (A = next_lde(A) ∧ B = next_lde(B) ∧ [[φ^lde]][A] → [[φ^lde]][B].

**Proposition 3.59.** Applying rule forcedDeleteEntityXfromY (referred to as ldde) in a well-formed VPM state preserves Invariant 3.38. Formally, ∀A, B : (A = next_ldde(A) ∧ [[φ^ldde]][A] → [[φ^ldde]][B].
As straightforward consequences, we state that these operations do exactly what we informally expected.

**Corollary 3.60.** Applying rule addEntityXtoY (referred to as add) with parameters \((X, Y)\) adds a corresponding entity \(X\) as a subcomponent to entity \(Y\). Formally, \(\forall \mathcal{A}, \mathcal{B} : \mathcal{B} = next_{add}(\mathcal{A}) \land (X \neq Y) \land \text{entity}(Y) \land \neg \text{componentOf}(X, Y) \rightarrow \text{componentOf}(X, Y)\) \(\mathcal{B}\) \(\mathcal{A}\).

**Corollary 3.61.** Applying rule softDeleteEntityXtoY (referred to as lsde) with parameters \((X, Y)\) removes the corresponding subcomponent entity \(X\) from entity \(Y\) provided that \(X\) is isolated in \(Y\). Moreover, if entity \(X\) is not contained by any other entity, it is removed. Formally, \(\forall \mathcal{A}, \mathcal{B} : \mathcal{B} = next_{lsde}(\mathcal{A}) \land (X \neq Y) \land \text{entity}(Y) \land \text{entity}(Y) \land \text{componentOf}(X, Y) \land \text{isXIsolatedInY}(X, Y)\) \(\mathcal{B}\) \(\mathcal{A}\) \(\rightarrow \neg \text{componentOf}(X, Y) \land (\exists Z : \text{entity}(Z) \land \text{componentOf}(X, Z)) \rightarrow \neg \text{entity}(X)\). \(\mathcal{B}\) \(\mathcal{A}\).

**Corollary 3.62.** Applying rule forcedDeleteEntityXtoY (referred to as lfde) with parameters \((X, Y)\) removes the corresponding subcomponent entity \(X\) from entity \(Y\). Moreover, if entity \(X\) is not contained by any other entity, it is removed. Formally, \(\forall \mathcal{A}, \mathcal{B} : \mathcal{B} = next_{lfde}(\mathcal{A}) \land (X \neq Y) \land \text{entity}(Y) \land \text{entity}(Y) \land \text{componentOf}(X, Y) \land \text{componentOf}(X, Y)\) \(\mathcal{B}\) \(\mathcal{A}\) \(\rightarrow \neg \text{componentOf}(X, Y) \land (\exists Z : \text{entity}(Z) \land \text{componentOf}(X, Z)) \rightarrow \neg \text{entity}(X)\). \(\mathcal{B}\) \(\mathcal{A}\).

Adding and removing connections (mappings)

Finally, we cope with the manipulation of connections and mappings. Since they are handled identically, only the creation and deletion of connections are specified in Algorithm 11 and 12, respectively.

### Algorithm 11 Adding connections globally / locally

```plaintext
rule addConnectionRfromAtoB(R, A, B) =
1: if \neg \text{connection}(R) \land \text{entity}(A) \land \text{entity}(B) \land \text{then}
2: \text{connection}(R) := \text{true};
3: \text{superType}(R, R) := \text{true};
4: \text{instanceOf}(R, R) := \text{true};
5: \text{from}(R) := A;
6: \text{to}(R) := B;
7: else \{\text{Precondition is false}\}
8: \text{skip}
9: end if

rule addConnectionRfromAtoBinE(R, A, B, E) =
10: if \exists E \neq E \land \text{entity}(E) \land \neg \text{componentOf}(R, E) \land \text{entity}(A) \land \text{componentOf}(A, E) \land \text{entity}(B) \land \text{componentOf}(B, E) \land \text{then} \{E is an entity not containing R; A and B are entities contained by E\}
11: if \neg \text{connection}(R) \land \text{then} \{\text{Connection R does not exist}\}
12: \text{addConnectionRfromAtoB}(R, A, B)
13: end if
14: \text{addComponentY}(R, E);
15: \text{checkConsistency}()
16: else \{\text{Precondition is false}\}
17: \text{skip}
18: end if
```

- **Global creation.** When adding a connection \(R\) globally to the VPM model space, the existence of the from and to entities, and the non-existence of connection \(R\) are required to be checked (Line 1 in Alg. 11). Then, the connection itself is created by setting the corresponding locations to true (Line 2–6 in Alg. 11).
• **Local creation.** In case of local addition of a connection $R$, we also need to check whether the entities are part of the same container entity $E$ (note the underlined literals in Line 10 of Alg. 11). Then a new connection is only created if connection $R$ is non-existent. In both cases, $R$ is added to entity $E$ as a subcomponent.

**Algorithm 12 Deleting connections globally / locally**

```plaintext
rule delConnectionRfromAtoB($R$, $A$, $B$) =
   if connection($R$) ∧ from($R$) = $A$ ∧ entity($A$) ∧ to($R$) = $B$ ∧ entity($B$) then
      deleteFromHierarchy($R$)
   connection($R$) := false;
   supertype($R$, $R$) := true;
   instanceOf($R$, $R$) := true;
   from($R$) := undef;
   to($R$) := undef
   checkConsistency()
  else
   skip
end if

rule delConnectionRfromAtoBinE($R$, $A$, $B$, $E$) =
   if connection($R$) ∧ from($R$) = $A$ ∧ entity($A$) ∧ to($R$) = $B$ ∧ entity($B$) ∧ entity($E$) ∧ componentOf($R$, $E$) ∧ componentOf($A$, $E$) ∧ componentOf($B$, $E$) then
      delXfromComponentZ($R$, $E$)
   if ∃ $Z$ : entity($Z$) ∧ componentOf($R$, $Z$) then {Connection $R$ is not contained by any elements but itself}
      delConnectionRfromAtoB($R$, $A$, $B$)
   end if
   checkConsistency()
else {Precondition is false}
   skip
end if
```

• **Global deletion.** In order to obtain a consistent global removal of a connection $R$, we check again if the connection to be removed is well-formed, i.e. connection $R$ both source and target entities ($A$ and $B$) are also existent (see Line 1 of Alg. 12). Then connection $R$ is completely removed from the refinement and containment hierarchy (Lines 2–7). Finally, rule `checkConsistency` is called to ensure the well-formedness of the refinement hierarchy.

• **Local deletion.** When performing a local delete operation on connection $R$ then (after checking the existence of the elements in Line 12) we first remove $R$ from the container entity $E$ (Line 13). Then if connection $R$ is no longer contained by any other entities (Line 14), we explicitly remove it from VPM by calling the global delete rule `delConnectionRfromAtoB`. Finally, rule `checkConsistency` is called again to ensure the well-formedness of the refinement hierarchy.

Finally, we present the propositions formalizing the correctness of local and global rules for manipulating connections.

**Proposition 3.63.** Applying rule `addConnectionRfromAtoB` (referred to as `gac`) in a well-formed VPM state preserves Invariant 3.38. Formally, $∀ \mathfrak{A}, \mathfrak{B} : \mathfrak{B} = \text{next} \_\text{gac}(\mathfrak{A}) ∧ \llbracket \varphi^{ge}_{3.38} \rrbracket^A_κ → \llbracket \varphi^{ge}_{3.38} \rrbracket^B_κ$.

Applying rule `addConnectionRfromAtoBinE` (referred to as `lac`) in a well-formed VPM state preserves Invariant 3.38. Formally, $∀ \mathfrak{A}, \mathfrak{B} : \mathfrak{B} = \text{next} \_\text{lac}(\mathfrak{A}) ∧ \llbracket \varphi^{le}_{3.38} \rrbracket^A_κ → \llbracket \varphi^{le}_{3.38} \rrbracket^B_κ$.

**Corollary 3.64.** Applying rule `addConnectionRfromAtoB` (`gac`) with parameters $(R, A, B)$ adds a corresponding connection $R$ that leads between entities $A$ and $B$. Formally, $∀ \mathfrak{A}, \mathfrak{B} : \mathfrak{B} = \text{next} \_\text{gac}(\mathfrak{A}) ∧ \llbracket \text{entity}(A) ∧ \text{entity}(A) ∧ ¬ \text{connection}(R) \rrbracket^B_κ → \llbracket \text{connection}(R) ∧ \text{from}(R) = A ∧ \text{to}(R) = B \rrbracket^B_κ$.
Proposition 3.65. Applying rule delConnectionRfromAtoB (referred to as \textit{gdc}) in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : (\mathcal{B} = \text{next}_{gdc}(\mathcal{A}) \lor \mathcal{B} = \text{next}_{del}(\mathcal{A})) \land \llbracket \mathcal{E} \rrbracket_C^{E} \land \llbracket \text{connection}(R) \rrbracket_C^{E} = \text{false} \). Applying rule delConnectionRfromAtoBinE (referred to as \textit{ldc}) in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : (\mathcal{B} = \text{next}_{ldc}(\mathcal{A}) \lor \mathcal{B} = \text{next}_{del}(\mathcal{A})) \land \llbracket \mathcal{E} \rrbracket_C^{E} \land \llbracket \text{connection}(R) \rrbracket_C^{E} \rightarrow \llbracket \mathcal{E} \rrbracket_C^{E} \). 

Corollary 3.66. Applying rule delConnectionRfromAtoB (\textit{gdc}) with parameters \((R, A, B)\) removes the corresponding connection \(R\) that led between entities \(A\) and \(B\). Formally, \( \forall \mathcal{A}, \mathcal{B} : \mathcal{B} = \text{next}_{gdc}(\mathcal{A}) \land \llbracket \text{entity}(A) \land \text{entity}(A) \land \text{connection}(R) \land \text{from}(R) = A \land \text{to}(R) = B \rrbracket_C^{E} \rightarrow \llbracket \text{connection}(R) \land \text{from}(R) = \text{undef} \land \text{to}(R) = \text{undef} \rrbracket_C^{E} \).

Example 3.67. Let us extend our VPM model space containing the metamodel of graphs (defined in Example 3.30) by adding a new entity BiGraph, and new instance-of and supertype relations.

1. First we call \textit{addEntity(X)} with \textit{BipGraph} as parameter to create the corresponding entity.
2. Then we call \textit{addYSupertypeOfX(BipGraph, Graph)} in order to create an inheritance relation from \textit{BipGraph} to \textit{Graph}. However, as \textit{isYSupertypeOfX(BipGraph, Graph)} is evaluated to false in Line 2 of Alg. 5 (e.g., because entity Node in Graph is not yet refined in BipGraph), nothing is changed in the VPM model space.
3. Finally, when we call \textit{addXInstanceOfY(BipGraph, Graph)}, the \textit{isXInstanceOfY(BipGraph, Graph)} call trivially evaluates to true, since the freshly created BipGraph is an empty model (therefore nothing can contradict the refinement axioms).

3.5 A Model-Level Algebraic Representation of VPM Models

3.5.1 Characteristics of a model-level representation

Traditionally (i.e., in the four layer MOF architecture), when a new model element is added to a certain metalevel, all models in a higher metalevel (i.e., the metamodel, the metametamodel, etc.) are kept constant. In other terms, the type level should be precisely defined prior to adding the first element on the instance level.

As we showed in Sec. 3.4, our VPM approach allows much more flexibility for the domain engineers as it carries out the maintenance of modeling languages and their instance models automatically.

However, for many existing applications and analysis tools, it is useful to provide an alternative model-level representation of VPM models. In fact, the term \textit{an alternate representation} is a bit misleading, as different VPM model spaces would yield different model-level representations. Thus we only define how to derive a model level algebraic representation of a VPM model space \textit{provided that all new elements that are added to metamodels during the evolution of a VPM model space are a priori known}. In other terms, we may introduce an arbitrary (finite) number of new elements into instance models, but all new elements added to metamodels have to be selected from a priori known finite set (containing initially existing and potentially existing metaclasses).

The mathematical explanation for this restriction is that the vocabulary of the model should be a priori fixed. This is easily fulfilled when using a meta-level representation; however, in the case of a model-level representation, different metaclasses yield different vocabularies.

Note that this is a real restriction for our sophisticated VPM approach, but it is almost immediately fulfilled in current modeling (and metamodeling) practice, since we typically analyze instance models supposing that the metamodel is already fixed. Thus new elements are only introduced to instance models but not to metamodels.

The basic practical explanation for the distinction between meta-level and model-level representations of VPM models is that the \textit{meta-level encoding} of VPM models primarily allows to build \textit{interpreters} for VPM (due to its generality), while the \textit{model-level representations} may provide primary means to construct \textit{compilers} for VPM models (yielding better performance during execution).
Example 3.68. In order to emphasize the difference between meta-level and model-level representation, let us consider two entities Node and n1 where n1 is an instance of Node.

- The meta-level representation uses the formulae entity(Node) = true, entity(n1) = true, and instanceOf(n1, Node) = true to express this relationship.
- The model-level representation introduces the function symbol node and therefore uses entity(Node) = true (the Node metaclass is an instance of Entity), entity(n1) = true, and node(n1) = true (n1 is an instance of Node and Entity as well) to express the same relationship.

Naturally, due to the inheritance hierarchy, the n1 could be set to true for several classes on the same metalevel which are superclasses of Node.

This example also emphasizes that the model-level representation of a VPM model includes the meta-level representation but we introduce new function symbols \( f_X \) for all model element \( X \) if they can be instantiated (thus excluding elements on the lowest metalevel).

Definition 3.69 ((Model-level) Vocabulary of VPM). The (meta-level) vocabulary \( \Sigma_{vpm}(M) \) of a VPM model \( M \) consists of \( \Sigma_{vpm} \) and the following characteristic function symbols:

- for all entities \( e \in M \): \( e/1 \)
- for all connections \( c \in M \): \( c/1 \)
- for all mappings \( m \in M \): \( m/1 \).

Definition 3.70 ((Model-level) State of VPM). The superuniverse \( |\mathcal{M}|_M \) of a state \( \mathcal{M}_M \) of a VPM model (i.e., of vocabulary \( \Sigma_{vpm}(M) \)) is identical to that of the VPM framework, i.e., it contains the identifiers \( id_P \) of all potential model elements \( P \).

The (model-level) interpretation of function symbols of \( \Sigma_{vpm}(M) \) is as described in Def. 3.29, and extended for model specific function symbols:

- **Entities**: if \( e \) is introduced for an entity \( E \), then 
  \[ e^\mathcal{M}(id_X) := \text{entity}^\mathcal{M}(id_X) \land \text{instanceOf}^\mathcal{M}(id_X, id_E); \]
- **Connections**: if \( c \) is introduced for a connection \( C \), then 
  \[ c^\mathcal{M}(id_X, id_A, id_B) := \text{connection}^\mathcal{M}(id_X) \land \text{instanceOf}^\mathcal{M}(id_X, id_C) \land \text{from}^\mathcal{M}(id_X, id_A) \land \text{to}^\mathcal{M}(id_X, id_B); \]
- **Mappings**: if \( m \) is introduced for a mapping \( M \), then 
  \[ m^\mathcal{M}(id_X, id_A, id_B) := \text{mapping}^\mathcal{M}(id_X) \land \text{instanceOf}^\mathcal{M}(id_X, id_M) \land \text{from}^\mathcal{M}(id_X, id_A) \land \text{to}^\mathcal{M}(id_X, id_B); \]

3.5.2 Model-level VPM operations

While we were able to define a set of operations that manipulate the meta-level encoding of VPM models, we may only define a such operations on the model level provided that the metamodel is already fixed. In other terms, in general, we may only define an operation scheme which should be adapted to the specific metamodel.

Basically, a create and a delete operation has to be defined for instantiating each entity, connection and mapping in the metamodel (in contrast to the meta-level case where only a single operation was defined for entities, connections and mappings). Using a UML analogy, we may add and remove classes in UML (by using meta-level operations), and now we are aiming at introducing such for specific stereotypes (on the model-level) that refine classes into subcategories.

In Alg. 13, we introduce such operations for instantiating a fictitious entity Class. In both cases (addEntityClass and delEntityClass), we first call the meta-level operations, and then we set the corresponding location of function symbol class.

Similarly, we may define a pair of operations (addConnectionAssoc and delConnectionAssoc in Alg. 14) for each association (connection) in the metamodel to create and remove instance of that association.

Now we could easily establish similar correctness theorems as before in Sec. 3.4 but this time we omit them for space considerations.
Algorithm 13 Model-level manipulation of entities (a scheme)

\textbf{rule} addEntityClass(\(E\)) =
1: addEntity(\(E\));
2: addXInstanceOfY(\(E, \text{class}\));
3: class(\(E\)) := \top
\textbf{rule} delEntityType(\(E\)) =
4: forcedDelEntity(\(E\));
5: class(\(E\)) := \bot

Algorithm 14 Model-level manipulation of connections (a scheme)

\textbf{rule} addConnectionAssoc(\(C, A, B\)) =
1: addConnectionRfromAtoB(\(C, A, B\));
2: addXInstanceOfY(\(C, \text{assoc}\));
3: assoc(\(C, A, B\)) := \top
\textbf{rule} delEntityType(\(C\)) =
4: delConnectionRfromAtoB(\(C, A, B\));
5: assoc(\(C, A, B\)) := \bot

3.5.3 A metamodel hierarchy

With this model-level representation, we obtain a metamodel hierarchy (see Table 3.4) that is closely related to the four-layer MOF architecture. Each model element appears as an instantiation of an element situated on a higher metalevel and a function symbol. Mathematically, we obtain a hierarchy of meta-representation of terms (for a precise mathematical treatment and meta-programming applications see [40]).

<table>
<thead>
<tr>
<th>Metalevels</th>
<th>New function symbols</th>
<th>Sample terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>top-most meta-level</td>
<td>entity</td>
<td>─</td>
</tr>
<tr>
<td>(top VPM model)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intermediate meta-level</td>
<td>class</td>
<td>entity(Class)</td>
</tr>
<tr>
<td>(MOF metamodel)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>meta-level</td>
<td>node</td>
<td>entity(Node)</td>
</tr>
<tr>
<td>(MOF metamodels)</td>
<td></td>
<td>class(Node)</td>
</tr>
<tr>
<td>model-level</td>
<td>─</td>
<td>entity(n1)</td>
</tr>
<tr>
<td>(MOF models)</td>
<td></td>
<td>node(n1)</td>
</tr>
</tbody>
</table>

Table 3.4. The model level algebraic representation of VPM models

1. On the top of the hierarchy, we find the top VPM elements, i.e., the metamodel of the VPM approach (see Fig. 2.11).
2. Unlike the MOF standard, the concepts of existing metamodeling approaches (such as the \textit{Class} concept of MOF itself) can be integrated into the model-level VPM representation on a lower, intermediate meta-level.
3. Constructs of new modeling languages (such as \textit{Node} of graphs) can be defined on the lowest metalevel. Note that new elements on this and all the above metalevels should be taken from an a priori known finite set (since the vocabulary is not allowed to be extended).
4. Finally, an arbitrary (finite) number of new model elements are allowed to be introduced on the instance level.

Notice that if this metamodel hierarchy is truncated to the lower two metalevels, we obtain the concepts of traditional MOF metamodels and instance models or type and instance graphs (see Sec. 2.2).
3.6 Conclusions

In the current chapter, I proposed a unified semantic framework for VPM metamodeling based upon the mathematical paradigm of abstract state machines (following [172,184]).

1. **Algebraic description of VPM models.** I proposed a meta-level (Sec. 3.3.1) and model-level (Sec. 3.5) algebraic representation of VPM models,
2. **Consistency criteria for the algebraic description of VPM models.** I introduced well-formedness constraints (in Sec. 3.3.2) in the form of invariants.
3. **Elementary VPM operations.** I proposed elementary operations that provenly guarantee the consistent manipulation of the algebraic representation of VPM models, i.e., that none of the invariants are violated when applying them in a consistent state.

**Conceptual relevance**

The conceptual importance of this chapter is threefold.

- First, the elementary operations of the chapter will provide a uniform and higher level unit for constructing an operational semantics for modeling languages and transformations.
- Furthermore, the algebraic representation of VPM models is an innovative use of abstract state machines for metamodeling purposes.
- Finally, all the correctness and completeness proofs throughout the current thesis will use the universal ASM formalism. The universality of ASMs is demonstrated in [75] where Gurevich claims that each algorithm can be formally captured by an appropriate (sequential) abstract state machine.

**Practical relevance**

The practical relevance of the chapter is a consequence of the fact that an extensive tool support (e.g., the ASM Workbench [54], or AsmL within the .Net framework [21]) is existent for constructing, formally analyzing (by testing [69] or model checking techniques [196]) and simulating ASM specifications. In this respect, we can easily implement the VPM metamodeling core as ASM specifications in order to support the uniform handling of modeling languages taken from arbitrary domains. Since efficient ASM engines are available, the elementary VPM operations proposed in the current chapter are directly applicable.

- Meta-level representations are appropriate for an interpreter-based implementation approach where a single ASM interpreter is sufficient to simulate a model parameterized by the operational semantics of its language. The main advantage of this meta-level approach is that model transformations accessing different meta-levels can be specified. A typical example can be transformation rules that generate further transformation rules as result (e.g., a set of rules that generate another set of construction rules from a metamodel to build well-formed model instances).
- Model-level representations are appropriate for a compiler-based implementation where individual ASM transformation programs are generated according the operational semantics of a language. The main advantage of a compiler-based approach is that its performance is typically much better than in the case of interpreters.

Moreover, as the virtual machine of several modern programming languages (like Prolog or Java) has been described in the ASM formalism, ASMs may easily provide a mathematically precise integration platform between various modeling techniques (such as UML or graph transformation) and implementation platforms (like programming languages) as well. Taking an ASM approach, provenly correct transformations from semi-formal or formal specifications into modern programming languages can be developed.

Building upon these ASM based foundations, in the upcoming chapter, the focus of the thesis is turned towards capturing the dynamic semantics of modeling languages and model transformations between them.
Visual Specification of Modeling Languages and Model Transformations

In the current chapter, I adapt the paradigm of graph transformation to simultaneously specify (i) the dynamic semantics of modeling languages defined in VPM, and (ii) model transformations between modeling languages. Moreover, I define a refinement calculus of graph transformation rules to enhance the reuse of dynamic behavior.

As a special emphasis will be put later on deriving different implementation and verification strategies for this high-level and visual specification method, we use ASMs as a common basis to define the semantics of graph transformation (together with theoretical variations) instead of the traditional category theoretic foundations.

4.1 Specifying Dynamic Behavior: An Overview

4.1.1 Dynamic semantics of models and modeling languages

As we already sketched in Sec. 2.1, the dynamic behavior of a modeling language can be specified either in an operational, denotational or axiomatic way.

- An **operational semantics** of a language defines how well-formed model instances evolve over time. In other terms, they define local transformations on models, which yield a set of potential execution traces. The semantics of imperative programming languages and model checkers frequently follow the operational approach.
- A **denotational semantics** of a language is a transformation into a precisely defined semantic domain (which is, in turn, another modeling language). In other terms, we define an interpretation that provides a meaning for each abstract concept in the language. The semantics of functional programming languages or traditional first order logic is defined in a denotational way.
- An **axiomatic semantics** of a language describes preconditions that must hold prior to executing an operation and postconditions that must hold after executing the operation.

From a practical (i.e., language/transformation designer’s) viewpoint, the operational and denotational way has a clear advantage over the axiomatic approach as the former two are directly executable. In other terms, we can easily implement an abstract machine that would directly simulate the potential behavior of models. On the other hand, as we demonstrate in Sec. 4.2, there is still not a final showdown between these approaches in practice.

According to a different categorization of dynamic semantics, we may distinguish between meta-level and model-level approaches. A short explanation of each strategies is given below.

- A **meta-level** semantics describes the dynamic behavior of any well-formed instance of a modeling language. In other terms, we have no a priori knowledge how the actual user model looks like which is controlled by our meta-level semantics. Note that the meta-level semantics of two different models of the same language are always the same.
4 Visual Specification of Modeling Languages and Model Transformations

- A **model-level** semantics specifies the dynamic behavior of a specific instance model of the language. In other terms, the model-level semantics of two different models of the same language may differ considerably. When defining a model-level semantics to a modeling language, we typically specify a semantic pattern (or schema) which is parameterized by the specific model instance in order to obtain a mathematically precise semantics.

Roughly spoken, meta-level approaches use (quantified) variables for generalization purposes while model-level approaches use parameters.

For **specifying** modeling languages, meta-level approaches typically supersede model-level ones as they are typically much more concise and more general. Naturally, this generality is a drawback from a **verification** viewpoint. For instance, model checker tools accept model-level (operational) specifications while meta-level approaches require, in general, sophisticated theorem proving techniques.

However, system designers are more interested in a simple, concise and graphical notation to specify their system or modeling language rather than in an efficient verification. Therefore, we may conclude that we should look for a meta-level specification technique for our purposes.

### 4.1.2 Specification of model transformations

Model transformation approaches that aim to define translation from the source modeling language into the target language can be grouped into two main categories:

- **Relational (declarative) approaches**: these approaches typically *declare a relationship* between elements of the source and target language. These solutions are frequently bi-directional in the sense that manipulations in the source model appear directly in the target model and vice versa.

- **Operational (or unidirectional) approaches**: these techniques *describe the process* of a model transformation from the source to the target language. As they aim at defining a mapping from the source to the target language, these approaches are typically uni-directional.

In order to unify the specification techniques of modeling languages and transformations between them, we may state that relational model transformation approaches are typically defined in an axiomatic or (in the best case) denotational way while operational model transformation approaches have (unsurprisingly) an operational semantics.

### 4.1.3 Problem statement

In **industrial** practice, the questions of **defining dynamic semantics to a modeling language** and **specifying model transformations between two languages** could be kept separately from each other. For instance, we may carry out a model transformation between two modeling languages without precisely defined dynamic semantics. Alternatively, we may use completely different techniques for these purposes.

However, according to the requirements of the current thesis (see 1.2.3), we should preferably adapt the same specification technique both for defining languages and transformations in order to allow transformation engineers to use a single methodology. As our primary focus is on precisely defining model transformations, we summarize existing model transformation approaches and investigate if there is a model transformation technique that is simultaneously applicable also to define the dynamic semantics of a modeling language.

### 4.2 Related results

Despite the fact that the problem of translating **textual** programming languages into each other has been thoroughly studied for several decades, when our current research (triggered by transformation
4.2 Related results

problems arisen in HIDE) was started, very few approaches (e.g., [148]) and tools (e.g., [93]) were existent to provide a solution for the model transformation problem (between different modeling languages) that fulfill the requirements of a UML environment. In fact, our initial proposal in [187, 188, 190] was one of the very first complex solutions in the field.

However, automated model transformations have recently become a hot topic in the MDA/UML environment thus a considerable amount of related work is available by now. This brief overview, which focuses on the comparison of metamodeling based model transformation approaches, summarizes and categorizes results presented in the International UML Conferences in the recent years, the Workshops on Transformations in UML (WTUML 01, 02), and related workshops and the 1st International Conference on graph transformation.

All approaches discussed below will be evaluated according to their (i) expressive power for capturing model transformations (including its visual nature), (ii) precise mathematical background, (iii) automated generation of an efficient implementation, and (iv) relatedness to industrial standards. We believe that these are the most important features for evaluating complex solutions for model transformations. As, basically, all the approaches are using metamodeling techniques for describing (the static structure of) modeling languages, this evaluation criterion is meaningless.

4.2.1 Relational approaches

Most common relational approaches [4, 5, 81, 96, 112] combine metamodeling with OCL constraints to define mappings between source and target languages. The semantics of a model language can therefore defined in a denotational way, i.e., by mapping it into precisely defined semantic domain.

**Relatedness to industrial standards:** [+] OCL is part of the UML standard [120, 162, 163],

**Mathematical preciseness:** [+] The mathematical preciseness of OCL is not provided in the standard but in extensive research such as [37]. We may conclude that OCL currently has, or soon will have a precise mathematical background.

**Automated implementation:** [-] UML CASE tools (e.g., [70]) and OCL libraries (e.g., [1]) exist for the implementation and run-time evaluation of OCL constraints. However, as such specification of model transformations are declarative, its not at all trivial how to automatically derive a transformation script that actually transform source and target models into each other.

**Expressiveness:** [+/-] Even though bi-directionality might be convenient for equivalence transformations (which are frequently rather simple and syntactic), transformations with deliberate loss of information (such as abstractions, projections, filtering) cannot be properly expressed in this way. Moreover, when metamodel based mapping should be further restricted by OCL constraints, they end up in a purely textual language (even though several attempts [34, 35, 141] aim at to visualize it).

Naturally, mappings between source and target modeling languages can be captured in tool specific notations as in [71] instead of OCL. The authors report a (semi-)automated generation of Prolog programs from their specification language. However, they are less related to industrial standards and expressiveness is still limited as before.

4.2.2 Operational approaches

As models can be uniformly represented in XMI, therefore, it is a straightforward idea to define model transformations using XSL (eXtensible Stylesheet Language) Transformations, which is part of the the XML standards as did, for instance, in [55, 136].

**Relatedness to industrial standards:** [+] XSLT is an industrial standard for describing syntactic transformations on XML documents.
Automated implementation: [-] There are various XSLT engines that execute XSLT transformation scripts. Unfortunately, according to our experiments (in several student projects) XSLT is very inefficient as a transformation language for complicated model transformations when models are not trees but complex graph structures.

Mathematical preciseness: [+/-] XSLT has a precise mathematical background, but this is not part of the standard.

Expressiveness: [-] XSLT is a textual language tailored to tree transformations, and almost impossible to use “manually” as a specification language for complex model transformations.

Model and metamodels can also be represented formally as algebraic specifications, and we can use rewriting logics [108] (or term rewriting) to capture the dynamic evolution of models as done, for instance, in the Maude framework [41,42]. As demonstrated in [195], term rewriting rules may also provide a means to define model transformation in an operational way.

Such transformations based on term rewriting typically have a limitation that the transformation process has to be **reductive**. In other terms, we start from a complex model (or a complex expression), and we are aiming to reduce it into a more simple form. Unfortunately, model transformations between two languages are typically **generative** since the target model has to be built up from scratch thus term rewriting-based solutions (and especially **tools**) can be insufficient.

Mathematical preciseness: [++] Rewriting logic [108] has a very rich semantic background. Maude also provides automated means for reasoning on models.

Automated implementation: [+] Maude has a very efficient rewriting engine (much faster than Prolog, for instance).

Expressiveness: [+-] While confirming its rich mathematical expressiveness, Maude is still a textual language. Moreover, its theoretically necessitated restriction imposed on the structure of rewriting rules requires sophisticated tricks to open the framework applicable to model transformations having a generative nature.

Relatedness to industrial standards: [+-] Maude in itself is very far from industrial standards, however, it is a common framework for specifying semantics for UML models [65,160,194].

Individual model transformations were designed in the RIVIERA framework [144], where UML models are transformed to the Maude language in order to carry out formal analysis and verification, however, it cannot really be considered as a general framework for designing transformations.

Transforming UML models into semantic domains to carry out formal verification characterizes the **UMLaut** [93] approach as well, however, they propose a general methodology for only transformations within UML.

### 4.2.3 Graph transformation based approaches

For the recent years, graph transformation has become the most popular specification paradigm for capturing the operational semantics of visual modeling languages, and their transformations as demonstrated by several approaches and tools discussed below in more details. Their “score sheet” in general is as follows.

**Expressiveness:** [++] Graph transformation is visual and (virtually) there are no restrictions for rules, therefore arbitrary model transformations can be described at a very high abstraction level.

**Mathematical preciseness:** [+] Graph transformation has a rich and well-founded theory summarized in [58,60,142].

**Automated implementation:** [+] A wide range tools are available that provide automation for transformations. Such tools include traditional graph transformation tools like AGG [64], PROGRES [151], DiaGen [99] or GenGED [14], and more recent ones embedding graph transformation
into a UML / metamodeling environment like FUJABA [114], ATOM3 [52], GME [107] or VIATRA [49,191].

**Relatedness to industrial standards:** [+/-] Graph transformation itself is not an industrial standard. However, on the one hand, it often complements metamodels and UML models, on the other hand, there are initiatives that aim at providing a standard XML description for graphs (GXL [150]) and graph transformation (GTXL [157]). Moreover, as demonstrated in [180,189], MOF metamodeling techniques are applicable to such document design via the XMI standard.

In the sequel, we give a brief comparison existing graph transformation based approaches and the concepts of VIATRA.

**The Paderborn Approach.** Undoubtedly, the most related results concerning transformations of UML models into semantic domains by graph transformation techniques were reported at Univ. of Paderborn in [61–63,84]. In fact, recent joint research [16,17] demonstrated that the two approaches are convergent.

The Paderborn approach propose a methodology for dynamic metamodeling and model transformations and relies on existing graph transformation tools (like AGG or FUJABA) as tool support. A major difference (in contrast to the VIATRA framework) is that the authors propose to use a special compound pair of graph transformation rules to manipulate source and target models. For their main feasibility study [62,84], UML models are transformed into CSP (process algebra) expressions to prove behavioral consistency of UML designs.

**The Vanderbilt ISIS Approach.** Related work has been carried out recently at the Vanderbilt University [153,154] to carry out transformations between metamodels, which transformations are also specified by graph transformation rules. The main specialty in their approach is that cardinality constraints can also be used in the patterns. Additionally, to improve the performance of the transformation process, the RHS of a preceding rule is glued together with the LHS of its subsequent rule, which serves as special control structures.

**Triple graph grammars.** As a general graph transformation approach tailored to language translations, triple graph grammars (TGG) [148] provide translations between modeling languages that are executable in both directions (source-to-target and target-to-source). However, bi-directional transformations are less expressive than uni-directional approaches, therefore certain structural limitations have to be imposed on TGG rules (such as no modifications are made on the source model) in order to increase their expressiveness.

### 4.2.4 Model transformation tools

**ATOM3.** ATOM3 [52] is a multi-paradigm visual modeling framework also using graph transformation for defining semantics of individual modeling languages and transformations [53,94]. In addition to discrete modeling languages (like Petri nets, statecharts, etc.) they also target integrating domains of continuous models (as widely used control theory).

Compared to VIATRA, transformations in ATOM3 are also directly based on graph transformation rules without explicit control structures, moreover, the entire framework is not as related (yet) to industrial standards as VIATRA is. On the other hand, ATOM3 provides a very rich simulation environment simultaneously for discrete and continuous models.

**FUJABA.** FUJABA [114] is UML CASE tool that has embedded (object-oriented) graph transformation facilities. Therefore, it may also serve as a very efficient model transformation engine between modeling languages defined by metamodeling techniques. In fact, the idea of using UML as the visual syntax for graph transformation rules and control structures first appeared in [66].
For that purpose, FUJABA uses (so-called) story diagrams [66], which merges activity and collaboration diagrams in order to combine graph transformation rules and control structures. VIATRA, on the other hand, uses stereotyped class diagrams for defining rules and statechart diagrams for representing control structures. We believe that while the choice of such UML representation in FUJABA is closer to the UML philosophy, UML representation used in VIATRA provides a closer correspondence to the special needs of model transformations.

Although VIATRA is not a general purpose graph transformation environment but rather tailored to the needs of industrial strength model transformations, we provide a brief comparison with the most commonly used traditional graph transformation tools.

PROGRES. PROGRES [151] is a meta-environment for rapid prototyping with a wide range of applications (ranging from database design to reverse engineering [47] to the design of CAD tools [156]). PROGRES has a very rich description language for specifying models and metamodels (based on graph schemata [149] and and combined graphical-textual constraints), and rules (using multiobjects, path expressions) and comes with a fast execution engine.

In fact, we may easily find the corresponding counterparts in PROGRES for the underlying modeling concepts in VIATRA. For instance, (i) derived relations can easily be implemented by PROGRES graph expressions, (ii) multiobjects and cardinality constraints may be used as a substitution for hierarchical patterns in VIATRA, and (iii) the available control structures are also related (except for formal semantics, which is unique in VIATRA when compared to all related graph transformation tools). Naturally, as VIATRA is tailored to the special needs of model transformations, handling multiple metamodels is more convenient thanks to reference graphs.

AGG and GenGED. AGG [64] is Java based graph transformation framework offering the unique feature of critical pair analysis to detect potential conflicts of rules. Recently, AGG has been extended to support the concepts of type graphs, which now provides metamodeling facilities to the environment.

Compared to VIATRA, AGG allows plain negative conditions for rules, while control structures (if required) can be implemented in Java through API calls. On the other hand, arbitrary Java classes can be used as attributes, which is a powerful feature.

AGG also serves as the underlying graph transformation framework for the visual modeling environment GenGED [14] which aims at the purely visual definition of arbitrary modeling languages, however, focusing on the concrete visual syntax rather than the abstract syntax in contrast to metamodeling techniques.

DiaGen. DiaGen [99] is a diagram editor generator used for automatically generating editors for visual modeling languages (like GenGED). What makes DiaGen unique from a graph transformation point of view is the use of hypergraphs (with edges leading between two edges) and hyperedge replacements as rules.

As a summary (although our evaluation scores for different approaches cannot really be objective), we notice that graph transformation is (at least) one of the strongest approaches from an engineering aspect to tackle the problem of model transformations in a MDA environment. However, even most graph transformation based approaches typically use a special mathematical construct (pair of graphs, triple graph grammars) to capture transformations between modeling languages. Moreover, for many practical cases, the level of non-determinism in graph transformation systems without control structures is too high.
4.2.5 Own contribution

In the current chapter, I adapt graph transformation to VPM metamodeling techniques in such a way that allows a visual but mathematically precise specification of the dynamic operational semantics of both modeling languages and model transformations.

- First in Sec. 4.3, I overview the mathematical paradigm of graph transformation in an informal way and I define a precise operational semantics for graph transformation based on abstract state machines and the elementary operations of Sec. 3.4.
- Then in Sec. 4.4, I define control structures tailored to the needs of specifying modeling languages and model transformations that restrict the process of graph transformation.
- In Sec. 4.5, I introduce the concept of rule refinement that allows the reuse of dynamic behavior in addition to the reuse of classes and metamodels (packages).
- Finally, Sec. 4.6 concludes the chapter summarizing the main theoretical and practical achievements.

The basic information flow of the chapter is summarized in Fig. 4.1.

4.3 An ASM Semantics for Graph Transformation

Graph transformation combines the advantages of graphs and rules into a single computational paradigm, frequently used for generation, manipulation, recognition, and evaluation of graphs by applying local modifications on them.

In the sequel, we define graph transformation rules and their application to a model instances informally. Afterwards, the precise semantics will be defined operationally based on ASMs and the algebraic representation of VPM models. Note that this is behaviorally equivalent with the traditional category theoretical approaches of graph transformation, namely, the double pushout (DPO) [45] and single pushout (SPO) [59] approach but provides a better theoretical fit to the rest of the thesis.

Also note that due to space considerations, we only cover how to derive a meta-level ASM representation with strict mathematical preciseness, since the derivation of a model-level ASM representation becomes rather intuitive afterwards.
4.3.1 Graph transformation rules

Definition 4.1 (Informal) Graph transformation rule. A graph transformation rule over a metamodel $MM$ is a 6-tuple $r = (Lhs, Neg, Rhs, Cond, Assign, par)$, where $Lhs$ is the left-hand side graph, $Rhs$ is the right-hand side graph, while $Neg$ denote the (optional) negative application condition graph(s) with the following restrictions.

- All graphs are well-formed instances of the metamodel $MM$.
- $par$ is a function that maps each graph in $Neg$ to another graph in $Neg$ or $Lhs$ imposing a tree on $Lhs$ and $Neg$ graphs with $Lhs$ as the root element in the tree.
- Each node in $Lhs$ and $Neg$ may contain additional conditions $Cond$, which define boolean functions (predicates) on attributes.
- Each node in $Rhs$ may have attribute assignments $Assign$ associated to them, which can be interpreted as attribute updates.

Graphs in a graph transformation rule may share certain nodes and edges with each other. More specifically, graph elements in the $Lhs$ may also appear in the $Rhs$ graph. Moreover, $Lhs$ and $Neg$ graphs are arranged into a strict tree hierarchy (as defined by the $Par$) with the additional constraint that graphs appearing in a child element in the tree may contain nodes and edges of graphs appearing in a parent element in the tree but not vice versa. This technically means that negative conditions can be embedded into each other resulting in double (or arbitrarily complex) negations.

In order to clarify the structure of graph transformation rules (especially in case of double negations) and our notational conventions used in the rest of the thesis, a sample rule $enableTransR$ taken from the Petri net semantics presented in Sec. 8.1 is depicted in Fig. 4.2.

![Graph Transformation Rule]

**Fig. 4.2.** Double negation in rules

Example 4.2 (A graph transformation rule with double negation). Rule $enableTransR$ of Fig. 4.2 is structured as follows.

- The $Lhs$ graph consists of a single node $T$ of type $Trans$.
- The negative application condition $Neg$ consists of two graphs $Neg1$ and $Neg2$ where $Neg2$ is a direct descendant of $Neg1$ in the tree. Negative application condition graphs are denoted by shaded areas in figures labeled with the NEG keyword.
  - $Neg1$ contains three nodes ($A$ of type $InArc$, $P$ of type $Place$ and $T$ of type $Trans$) and two edges of type $toTr$ and $fromPl$ leading from $A$ to $T$ and $P$, respectively. Note that node $T$ has to be implicitly contained by $Neg1$ (although it is not explicitly depicted in the figure) since a negative condition in itself should be a well-formed graph.
  - $Neg2$ contains nodes $K$ of type $Token$ and $P$ and an edge of type $tokens$ leading from $P$ to $K$ for the same reasons.
- The $Rhs$ graph contains a single node $T$ shared with $Lhs$. 
- The rule contains an attribute condition for checking the \textit{enable} attribute and an assignment for updating the same attribute.
- The function \textit{par} is defined as \( \text{par(Neg1)} = \text{Lhs} \) and \( \text{par(Neg2)} = \text{Neg1} \).

Note that the distinction between model and rule elements is irrelevant from a modeling point of view as both are well-formed instances of the metamodel. However, we will keep the notions of rules and models separately at several parts in the thesis to improve legibility.

Finally, below we summarize the notations of MOF metamodels, models and rules.
- For the \textit{metamodel elements}, we use the terms \textit{classes}, \textit{associations}, and \textit{attributes} (also instead of the notions of node, edge and attribute types) in their traditional sense.
- For \textit{model elements} (elements of a user model on the instance-level), the terminology of \textit{objects}, \textit{links}, and \textit{slots} is used.
- Finally, for \textit{rule elements} (i.e., instance-level contents of a graph transformation rule) we use the terms \textit{nodes}, \textit{edges}, and \textit{slots/attributes}.

![Fig. 4.3. Metamodel of graphs, models and metamodels (meta-metamodel)](image)

A visual overview of our model and rule terminology is provided by the MOF metamodel of Fig. 4.3. Note that there is an implicit inheritance relation between \textit{Element} and all the other metaclasses, which is not shown explicitly to improve clarity.

**Graph transformation rules in VPM**

Due to the unifying nature of VPM (such as multiple containment of elements, dynamic refinement relations), the formal VPM representation of graph transformation rules is rather straightforward. We recommend to recall Def. 2.17.

**Definition 4.3 (VPM graph transformation rules).** A VPM graph transformation rule is a triple \( r = (\text{Pre}, \text{Post}, \text{par}) \), where \( \text{Pre} \) is a set of VPM patterns (called the \textit{precondition}), \( \text{Post} \) (or \( \text{Rhs} \) called the \textit{postcondition}) is a VPM pattern, and \( \text{par} : \text{Pre} \to \text{Pre} \) is a function imposing a tree structure on \( \text{Pre} \), i.e.,

1. \( \exists x, y : \text{par}(x) = y \land \text{par}(y) = x \) (\( \text{par} \) is acyclic)
2. \( \exists \text{Lhs} : \text{par}(\text{Lhs}) = \text{undef} \) (there is a unique root element \( \text{Lhs} \), which is the LHS of the rule)
3. \( \forall x : x \neq \text{Lhs} \to \exists y : \text{par}(x) = y \) (i.e., \( \text{par} \) is a single component).

Note that since mappings (attributes) are represented as edges between entities (similarly to connections), attribute conditions and assignments are identical to graph conditions / rewriting. Moreover, as the same VPM element can be contained by multiple entities, the informal well-formedness constraints of Def. 4.3.1 are automatically fulfilled.
Example 4.4. The VPM representation of enableTransR of Fig. 4.2 is depicted in Fig. 4.4.

Here $Pre = \{LHS, \text{Neg1}, \text{Neg2}\}$, and $Post = \text{RHS}$ where all $LHS$, $\text{Neg1}$,$\text{Neg2}$, and $\text{RHS}$ are entities. Since metalevels are treated uniformly in VPM, each of them may contain elements from any metalevels. For instance, in $LHS$, entity $Trans$ is a meta-level element while entity $T$ is a model-level instance of $Trans$.

This time connections are depicted as arrows with a black arrowhead (see $\text{tokens}$, for instance) while mappings are depicted with an open arrowhead (such as $enable$) in order to better emphasize the fact that a VPM model is a (hierarchical) graph structure.

To highlight the difference between constants and variables in VPM graph patterns, we can say that, for instance, entity $trans$ and mapping $enable$ is a constant while entity $T$ and mapping $A1$ is a variable in graph pattern $LHS$.

4.3.2 Graph pattern matching

Now we discuss how an elementary step defined by a graph transformation rule is executed on a given instance model. As an initial step, the notion of graph pattern matching is defined.

Definition 4.5 (graph pattern matching (informal)). Let $G$ be the $Lhs$ graph or a graph in $Neg$. $G$ can be successfully matched to a model $M$ (i.e., a well-formed instance of its metamodel $M_M$) if and only if:

- **Positive pattern.** There exists an (isomorphic or non-isomorphic) image of graph $G$ in the model $M$, in other terms:
  - each node (of the graph) can be mapped to a type conforming object (of the model);
  - each edge can be mapped to a type conforming, and source-target preserving link;
  - each graph element should be mapped to different model elements in case of isomorphic pattern matching;
  - all attribute conditions attached to graph nodes are satisfied by the corresponding slot values in the image;

- **Negative (prohibited) pattern.** There are not any successful matches for any graphs $G_{neg}$ in $Neg$ directly contained by $G$, i.e., such that $par(G_{neg}) = G$. When calculating these matchings, the mappings of shared nodes and edges that have already been matched in parent graphs are naturally preserved (and possibly extended).

We refer to the mappings that constitutes a pattern matching as “o”.

In theory, isomorphic or non-isomorphic pattern matchings can be prescribed separately for each graph transformation rule; however, in practice, graph transformation approaches and tools typically follow either the isomorphic or non-isomorphic case. The difference between isomorphic and non-isomorphic pattern matchings will be discussed later on in Example 4.17.
4.3 An ASM Semantics for Graph Transformation

Example 4.6 (Matching complex graph patterns). Successful and unsuccessful graph pattern matchings for the Lhs graph of rule enableTransR are demonstrated in Fig. 4.5.

- Transition t2 in model M1 can be successfully matched to node T1 in the pattern as the there are no incoming toTr links in the model thus the pattern matching of Neg1 is directly violated, moreover, the attribute condition enable = F is satisfied as well.
- Transition t2 in model M2 cannot be matched to node T1 in the pattern because the attribute condition enable = F is violated since the slot enable at object t2 is equal to T.
- Transition t3 of model M3 cannot be matched to node T1 in the pattern because Neg1 can be successfully matched to the model (with node mappings (A,a3) and (P1,p3)) as the matching of the negative subcondition Neg2 fails due to the lack of Token objects connected to p3.
- Finally, transition t4 of model M4 can be matched to node T1 in the pattern because the match of Neg1 fails due to the successful matching of its negative subcondition Neg2 with the mapping (K,k4). In this respect, the successful matching of Neg1 is discarded by the successful matching of Neg2, which in turn results in a successful match of the Lhs pattern (double negation).

In general, negative patterns can be embedded into each other in an arbitrary depth. However, for all practical cases (at least for all case studies presented in the current thesis), this pattern hierarchy is limited to a depth of two, i.e., a positive Lhs pattern potentially with a negation pattern Neg1 containing a negative subpattern Neg2. This defines double negation, which can sometimes be difficult to be understood by systems engineers. An equivalent interpretation of such two level patterns is that for all occurrences of Lhs, the successful matching (in other terms extension) of Lhs to Neg1 implies that the pattern Lhs ∪ Neg1 can be extended to Neg2 as well.

In order to easily navigate on the “troubled waters” of hierarchical negative patterns of arbitrary depth, the reader is recommended to use the following rule of thumb.

Remark 4.7 (Rule of thumb for interpreting negative patterns). In order to succeed with the matching of the root Lhs pattern, patterns appearing on an even level in the Par tree (like Lhs and Neg2 in the example) prescribe positive conditions that has to be matched if the matching of the parent pattern was previously successful.

On the other hand, patterns appearing on an odd level in the tree (like Neg1) prescribe negative conditions that (i) either must not be matched, or (ii) if a successful matching was found, all the child patterns should be successfully matched as well.

Graph pattern matching in VPM

As for the formal definition of pattern matching, we can first derive an ASM formula from a pattern. Then basically, we should (i) check in the VPM model instance for the existence (or non-existence) of constant elements of the pattern, and (ii) assign a corresponding element of the instance VPM model to each variable in the pattern. After that negative subpatterns (direct descendents in the pattern tree) are turned into negated subformulas with existentially quantified variables.
Definition 4.8 (ASM representation of a VPM pattern). Let \( P \) be a VPM pattern with variables \( \text{var}(P) = X = \langle X_1, \ldots, X_n \rangle \). The ASM representation of a pattern \( P \) is a formula \( \phi_P(X_P) = \bigwedge_i \phi_i(X_i^P) \) where \( \phi_i(X_i^P) \) is the formula (with \( k \) variables) defined in Table 4.1 according to Def. 3.29.

**Notation:** if \( a \) is a variable VPM element in a pattern \( P \) then \( A = X_a \); otherwise, if \( a \) is a constant then \( A = a.id \).

<table>
<thead>
<tr>
<th>abbrev</th>
<th>VPM (syntax)</th>
<th>ASM (semantics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi_a(X_a) ) where ( k =</td>
<td>\text{var}({e})</td>
<td>e \in P )</td>
</tr>
<tr>
<td>( \phi_r(X_r) ) where ( k =</td>
<td>\text{var}({r, a, b})</td>
<td>r(a, b) \in P )</td>
</tr>
<tr>
<td>( \phi_f(X_f) ) where ( k =</td>
<td>\text{var}({f, a, b})</td>
<td>f(a, b) \in P )</td>
</tr>
<tr>
<td>( \phi_{sup}(X_{sup}) ) where ( k =</td>
<td>\text{var}({a, b})</td>
<td>a \to b \in P )</td>
</tr>
<tr>
<td>( \phi_{inst}(X_{inst}) ) where ( k =</td>
<td>\text{var}({a, b})</td>
<td>a \leftrightarrow b \in P )</td>
</tr>
<tr>
<td>( \phi_{com}(X_{com}) ) where ( k =</td>
<td>\text{var}({a, b})</td>
<td>\exists : a = b[i] \in P )</td>
</tr>
</tbody>
</table>

Table 4.1. ASM encoding of VPM patterns

Example 4.9. The (meta-level) ASM representation of the LHS of rule \( \text{enableTransR} \) (of Fig. 4.2) is
\[
\phi_{LHS}(T, A) = \text{entity}(\text{trans}) \land \text{entity}(\text{bool}) \land \text{mapping}(\text{enable}) \land \text{trans} \land \text{to}(\text{enable}) = \text{bool} \land \text{entity}(T) \land \text{entity}(\text{false}) \land \text{mapping}(A) \land \text{from}(A) = T \land \text{to}(A) = \text{false} \land \text{instanceOf}(T, \text{trans}) \land \text{instanceOf}(A, \text{enable}) \land \text{instanceOf}(\text{bool}, \text{false}).
\]

In case of MOF models, we may alternatively use the model-level ASM representation of a VPM pattern, which abbreviates instance-of relations.

Example 4.10. The model-level ASM representation of the LHS of rule \( \text{enableTransR} \) is
\[
\phi_{LHS}(T, A) = \text{trans}(T) \land \text{bool}(\text{false}) \land \text{enable}(A1) \land \text{from}(A1) = T \land \text{to}(A1) = \text{false}.
\]

Definition 4.11 (ASM representation of preconditions of a rule). Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as the root element of \( \text{Pre} \)). The ASM representation of the precondition prescribed by the pattern tree \( \langle \text{Pre}, \text{par} \rangle \) is \( \phi_{Lhs}(X_{Lhs}) = \phi_{Lhs}(X_{Lhs}) \land (\forall a : \text{par}(a) = \text{root}) \land (\forall b : \text{par}(b) = a) \land (\forall X_p \text{ denotes the variables of pattern } p) \land (\forall X'_p \text{ denotes the new variables of pattern } p, \text{ i.e., those that do not appear in any ancestor pattern of } p)

\[
(\phi_a(X_a) \land (\forall X'_p : (\phi_b(X'_b) \land \ldots)))) \text{ where } X_p \text{ denotes the variables of pattern } p, \text{ and } X'_p \text{ denotes the new variables of pattern } p, \text{ i.e., those that do not appear in any ancestor pattern of } p.
\]

Example 4.12. The schematic ASM representation of precondition of rule \( \text{enableTransR} \) is \( \phi_{Lhs}(T, A1) = \phi_{Lhs}(T, A1) \land \exists A, P, C1, C2 : (\phi_{Neg}(T, A, P, C1, C2) \land \exists K, C3 : \phi_{Neg}(P, K, C3)).
\]

Note, for instance, that since \( T \) already appears in \( \text{Lhs} \) (as \( \phi_{Lhs} \) equals to Example 4.9), therefore it is not (re-)quantified when creating the formula of \( \text{Neg} \).

Definition 4.13 (Graph pattern matching). Given a state \( \mathcal{A}_M \) of a VPM model \( M \), and a formula \( \phi_P(X_P) \) derived from a pattern \( P \) (or pattern tree \( \langle \text{Pre}, \text{par} \rangle \) with root \( \text{Lhs} \)), the **matching** of \( \phi_P(X_P) \) in state \( \mathcal{A}_M \) is a variable assignment \( \zeta_i = \text{true} \) such that \( \llbracket \phi_P(X_P) \rrbracket_{\zeta_i}^{\mathcal{A}_M} \).
Example 4.14. A matching of the \(<\text{Pre,par}>\) pattern tree of rule \texttt{enableTransR} on VPM model \(M_1\) of Fig. 4.5 is an assignment \(\zeta\{T \mapsto t1\}\).

4.3.3 Application of a rule: An informal introduction

Based on the notion of graph pattern matching, we can easily define the process of a single application of a graph transformation rule.

Definition 4.15 (Rule application, transformation step – informal). The application of a graph transformation rule \(r\) to a model \(M\) (in other terms, a transformation step) rewrites the model (denoted as \(M^{r(o)}\)) by replacing the image of the pattern defined by \(Lhs\) pattern (restricted by prohibited subgraphs of \(Neg\) and attribute conditions \(Cond\)) with an image of the \(Rhs\) pattern. This is performed as follows.

1. Find a matching (occurrence) \(o\) for \(Lhs\) in model \(M\). This step also includes the checking of negative application conditions and attribute conditions as discussed above. In a general term, we check the precondition of a rule.
2. Remove. A part of the model \(M\) that can be mapped to the \(Lhs\) but not to the \(Rhs\) graph is then removed (yielding the context model/graph).
3. Glue. The image of the \(Rhs\) and the context model are glued together (by adding new objects and links as images of rule objects which can be mapped only to \(Rhs\) but not to the \(Lhs\) graph) to obtain the derived model \(M'\). Generally speaking, the \(Rhs\) defines the postcondition of a rule.

When a rule application prescribes the deletion of an object (i.e., there is a node in the \(Lhs\) but not in the \(Rhs\)), dangling links (edges) may remain, which would result in an ill-formed model. Various graph transformation approaches tackle this problem differently:

- In the single pushout approach (SPO) [59], all dangling links are implicitly removed from the model (implicitly in the sense that they are not explicitly defined by the rule itself).
- In the double pushout approach (DPO) [45], rule applications resulting in a model with dangling edges are forbidden (thus a rule defines exactly the modifications on the model).

For the moment, we do not fix these semantic questions, i.e., the concrete graph transformation approach will be a “parameter” for several results in this thesis.

As graph transformation is computationally (Turing) complete, we can safely propose to use graph transformation rules to define the dynamic behavior of instance models of arbitrary modeling languages. Graph transformation offers an operational specification technique to visually capture dynamic behavior.

Example 4.16 (Operational semantics of finite automata). The dynamic operational semantics of finite automata (which should capture how the current state of an automaton is changed when a transition is fired) is defined by the graph transformation rule \texttt{stepFA} in (the uppermost part of) Fig. 4.6.

The rule prescribes that if state \(S1\) of automaton \(A1\) is marked as current, and there is a transition \(T\) leading from \(S1\) to state \(S2\) then the current state of the automaton can be rewritten to \(S2\) as a result of the rule application.

More specifically, if we apply rule \texttt{stepFA} to finite automaton model of Fig. 2.3, the process of rule application will proceed as follows. We suppose that this model has already been initialized by adding a current link for all the states marked as initial (depicted on the concrete syntax by a striped circle for the current states).

1. One may find a successful matching of the LHS with the mappings \(\{A1 \mapsto a1; S1 \mapsto s1; S2 \mapsto s2; T \mapsto t1\}\) (which matching is highlighted in the first model of Fig. 4.6).
Fig. 4.6. Operational semantics for finite automata
2. Note that a graph transformation step may be non-deterministic. Thus another successful matching of the LHS may be found with the mappings \( A1 \rightarrow a1; S1 \rightarrow s1; S2 \rightarrow s3; T \rightarrow t3 \) (which matching is highlighted in the second model).

3. If we continue along this second matching, the current link leading between \( a1 \) and \( s1 \) should be removed according to the rule, which step yields the intermediate context model shown in the third model.

4. Finally, a new current link is created that leads from \( a1 \) to \( s3 \), and we obtain one possible result of the graph transformation step in the fourth model.

Example 4.17 (Isomorphic vs. non-isomorphic matchings). In general, a finite automaton may contain self loops as transitions, i.e., transitions that are leading from and leading to the same state. Therefore, such a self-loop transition should be fireable as well. If we prescribe isomorphic pattern matching for the previous rule in Fig. 4.6, when all rule elements have to be mapped into different model elements, self-loop transitions cannot be matched any more. Therefore, the previous transformation rule fulfills our informal expectations only if applied with non-isomorphic pattern matching.

Alternatively, if we stick to isomorphic pattern-matching, an additional transformation rules (depicted in Fig. 4.7) is required that handles the situation of self loops. In fact, this rule is side-effect free since a single current link is allowed to lead from (the image of) automaton \( A1 \) to state \( s1 \), therefore no rewriting takes places. But, at least, it handles (i.e., visits) the pattern of self-loops and thus would be able to modify the model if required.

![Fig. 4.7. Additional rule required for isomorphic pattern matching](image)

Now we define a formal semantics of applying VPM graph transformation rules by specially structured ASM rules.

### 4.3.4 ASM semantics for rule application

The basic structure of the ASM semantics of rule application can be divided into a selection, a deletion and an addition phase.

- **During the selection phase**, the variables of the root \( Lhs \) entity are instantiated by using the non-deterministic choose construct. We may also need to check whether an isomorphic pattern matching is prescribed.

- **Then in the deletion phase**, locations related to VPM model elements that appear only in the LHS of the rule but not in the RHS are falsified. Note that during this phase, we need to properly handle the potential dangling edges.

- **Finally, in the addition phase**, locations related to VPM model elements that appear only in the RHS of the rule but not in the LHS are set to true.
Identification condition

During selection, we may constrain (as done in the DPO approach [45] for nodes to be deleted) that two entities in the pattern graph are not allowed to have a unique image in the VPM model by an additional clause that prescribes that the values of the corresponding variables must not be equal.

Definition 4.18 (Identified entities). Let \( (\text{Pre}, \text{par}) \) be a pattern tree and \( Y \) an entity (called identified entity) in pattern \( P \in \text{Pre} \), which is not allowed to be matched to the same image as any other entity \( X_i \) of \( P \). Then the corresponding formula \( \phi_P(X_P) \) should be extended with the literal \( \bigwedge_{X_i} (X \neq X_i) \).

Note that by this liberal interpretation of isomorphic pattern matching we can uniformly handle both the traditional SPO (where non-isomorphic patterns are allowed) and the DPO approaches (where nodes that are removed by the rule should have a unique image).

Definition 4.19 (Identification condition). Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as the root element of \( \text{Pre} \)) applied in a VPM model state \( \mathfrak{A} \) to a matching pattern given by the variable assignment \( \zeta = \zeta_{\text{match}} \). The identification condition is the one where all entities that are removed by the rule are identified entities. Formally, \( \vartheta_r = \bigwedge_{e_i, e_j \in \text{Lhs} \land e_i, e_j \notin \text{Post} \land E_i = X_{e_i} \land E_j = X_{e_j}} \).

Deletion of elements, Dangling condition

The deletion phase is relatively simple, since we need to call the corresponding operation defined earlier in Sec. 3.4 for all VPM model elements that are in the image of the LHS but not in the image of the RHS. However, during deletion of entities, we need to precisely tackle the problem of dangling links (edges). Here, typically two approaches are followed: (i) they are either removed (yielding side-effects of the rule application), or (ii) the application of the rule should be forbidden. While we already handled the first case by forced delete operations in Sec. 3.4 (i.e., by additional implicit updates falsifying the proper locations), additional literals are required (as a dangling condition) in the pattern formula for the second case.

Definition 4.20 (dangling condition).

Informally, the dangling condition formula expresses that whenever a rule prescribes the deletion of an entity, then all related edges (i.e., connections or mappings) should be explicitly deleted as well by the rule.

Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as the root element of \( \text{Pre} \)) applied in a VPM model state \( \mathfrak{A} \) to a matching pattern given by the variable assignment \( \zeta = \zeta_{\text{match}} \). The dangling condition of the rule (in case of soft delete) is a formula defined in Table 4.2.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>ASM condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \delta_{e,f} )</td>
<td>if ( e \in (\text{Lhs} \setminus \text{Post}) ) and ( f_1, \ldots, f_n ) are all the mappings in ( \text{Lhs} \setminus \text{Post} ) then ( \delta_{e,f} : \text{mapping}(F) \land (\text{from}(F) = X_{e_i}) \land \text{to}(F) = X_{e_j} \land F \neq X_{f_1} \land \ldots \land F \neq X_{f_n} )</td>
</tr>
<tr>
<td>( \delta_{e,r} )</td>
<td>if ( e \in (\text{Lhs} \setminus \text{Post}) ) and ( r_1, \ldots, r_n ) are all the connections in ( \text{Lhs} \setminus \text{Post} ) then ( \delta_{e,r} : \text{connection}(R) \land (\text{from}(R) = X_{e_i}) \land \text{to}(R) = X_{e_j} \land R \neq X_{r_1} \land \ldots \land R \neq X_{r_n} )</td>
</tr>
</tbody>
</table>

Table 4.2. VPM dangling edge condition is ASM

Therefore the ASM representation of rule \( r \) that respects the dangling condition as well is \( \phi_{\text{Lhs}}^{\text{dang}} = \phi_{\text{Lhs}} \land \delta_r \) where \( \delta_r = \bigwedge_{e \in (\text{Lhs} \setminus \text{Post})} (\delta_{e,f} \land \delta_{e,r}) \).
Definition 4.21 (Deletion of elements). Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as the root element of \( \text{Pre} \)) applied in a VPM model state \( \mathcal{A} \) to a matching pattern given by the variable assignment \( \zeta \). The deletion of elements prescribed by \( \text{Lhs} \setminus \text{Post} \) is defined in Table 4.3.

<table>
<thead>
<tr>
<th>Abbrev</th>
<th>VPM condition</th>
<th>Abbrev</th>
<th>ASM statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \psi_{f_{j}} )</td>
<td>if ( e \in \text{Lhs} \land e \notin \text{Post} )</td>
<td>( \text{del}<em>{f</em>{j}} )</td>
<td>softDeleteEntity(( E ))</td>
</tr>
<tr>
<td>( \psi_{f_{c}} )</td>
<td>if ( e \in \text{Lhs} \land e \notin \text{Post} )</td>
<td>( \text{del}<em>{f</em>{c}} )</td>
<td>forcedDeleteEntity(( E ))</td>
</tr>
<tr>
<td>( \psi_{f} )</td>
<td>if ( f(a, b) \in \text{Lhs} \land f(a, b) \notin \text{Post} )</td>
<td>( \text{del}_{f} )</td>
<td>delMappingFromAtoB(( F, A, B ))</td>
</tr>
<tr>
<td>( \psi_{r} )</td>
<td>if ( r(a, b) \in \text{Lhs} \land r(a, b) \notin \text{Post} )</td>
<td>( \text{del}_{r} )</td>
<td>delConnectionFromAtoB(( R, A, B ))</td>
</tr>
<tr>
<td>( \psi_{\text{sup}} )</td>
<td>if ( (a \rightarrow b) \in \text{Lhs} \land (a \rightarrow b) \notin \text{Post} )</td>
<td>( \text{del}_{\text{sup}} )</td>
<td>delZFromSupertypeOF(( A, B ))</td>
</tr>
<tr>
<td>( \psi_{\text{inst}} )</td>
<td>if ( (a \rightarrow b) \in \text{Lhs} \land (a \rightarrow b) \notin \text{Post} )</td>
<td>( \text{del}_{\text{inst}} )</td>
<td>delZFromInstanceOF(( A, B ))</td>
</tr>
<tr>
<td>( \psi_{\text{comp}} )</td>
<td>if ( \exists : a = b[i] \in \text{Lhs} \land (\exists : a = b[i] \notin \text{Post} )</td>
<td>( \text{del}_{\text{comp}} )</td>
<td>delZFromComponentOF(( A, B ))</td>
</tr>
</tbody>
</table>

Table 4.3. ASM statements for deletions prescribed by a rule.

Note that when removing entities our choice of ASM call is dependent on the graph transformation approach we follow, i.e., the softDeleteEntity operation is only allowed to be called if the dangling condition was present during the pattern matching phase and vice versa.

The corresponding ASM rules that carry out the required deletions (in case of the SPO and DPO approach) is the following:

- **rule** deleter\(_{\psi_{f_{j}}}(X_{\text{del}}) = \text{seq} \ del_{\text{sup}}; \ \text{seq} \ del_{\text{inst}}; \ \text{seq} \ del_{\text{comp}}; \ \text{seq} \ del_{f}; \ \text{seq} \ del_{r}; \ \text{seq} \ del_{f_{c}} \ \text{where} \ X_{\text{del}} = \langle X_{q_{1}}, \ldots X_{q_{i}} \rangle \ \text{and all} \ q_{i} \ \text{such that} \ q_{i} \in \text{Lhs} \land q_{i} \notin \text{Post} \ \text{are variables.}**

- **rule** deleter\(_{\psi_{f_{j}}}(X_{\text{del}}) = \text{seq} \ del_{\text{sup}}; \ \text{seq} \ del_{\text{inst}}; \ \text{seq} \ del_{\text{comp}}; \ \text{seq} \ del_{f}; \ \text{seq} \ del_{r}; \ \text{seq} \ del_{f_{c}} \ \text{where} \ X_{\text{del}} = \langle X_{q_{1}}, \ldots X_{q_{i}} \rangle \ \text{and all} \ q_{i} \ \text{such that} \ q_{i} \in \text{Lhs} \land q_{i} \notin \text{Post} \ \text{are variables.}**

Let \( \text{Del}_{r} \) denote the set of elements that are explicitly removed by rule \( r \), i.e., \( \forall q : \psi_{q} \rightarrow Q \in \text{Del}_{r} \) (where \( \psi_{q} \) is still derived according to Table 4.3)

Note that since all variables in \( \text{Lhs} \) have a corresponding assignment in \( \zeta \) after a successful pattern matching, all parameters passed to elementary VPM operations are constants (e.g., \( E = e.id \) or \( E = X_{e} \) with \( \{X_{e} \mapsto e.id\} \in \zeta \)).

As a summary, we establish that our construction (up to now) is sound, i.e., whenever a graph transformation rule is successfully matched to a VPM model and the rule prescribes the deletion of some elements then those elements are successfully removed and the (intermediate) result is a consistent VPM model.

**Proposition 4.22.** Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as the root element of \( \text{Pre} \)) applied in a consistent VPM model state \( \mathcal{A} \) to a matching pattern given by the variable assignment \( \zeta \) (i.e., \( [\varphi_{(\text{Pre}, \text{par})}]_{\zeta}^{\mathcal{A}} = \text{true} \)). Then for all \( \mathcal{B} = \text{next}_{\text{delete}}(\mathcal{A}, \zeta) \),

1. \( \mathcal{B} \) is consistent
2. all elements \( q \) in \( \text{Lhs} \setminus \text{Post} \) that are matched in \( \mathcal{A} \) are successfully removed, i.e.,
   a) \( \forall e \in \text{Lhs} \land e \notin \text{Post} : [\text{entity}(E)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   b) \( \forall c \in \text{Lhs} \land c \notin \text{Post} : [\text{connection}(C)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   c) \( \forall m \in \text{Lhs} \land m \notin \text{Post} : [\text{mapping}(M)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   d) \( \forall a \rightarrow b \in \text{Lhs} \land a \rightarrow b \notin \text{Post} : [\text{supertype}(A, B)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   e) \( \forall a \rightarrow b \in \text{Lhs} \land a \rightarrow b \notin \text{Post} : [\text{instanceOf}(A, B)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   f) \( \forall a = b[i] \in \text{Lhs} \land a = b[i] \notin \text{Post} : [\text{componentOf}(A, B)]_{\zeta}^{\mathcal{B}} = \text{false} \).
As an abbreviation we write $\forall q \in Lhs \land q \not\in Post : \llbracket-\text{element}(Q)\rrbracket^\zeta_Q$.

3. If the dangling condition is also prescribed (i.e., $\llbracket \varphi_{\text{Pre}, \text{par}}^\text{Dang} \rrbracket^\zeta = \text{true}$) and no forcedDeleteEntity operations are used then all graph element $x$ (i.e., entity, connection or mapping) that was removed as a result of rule application should be explicitly deleted by $r$. Formally,$a) \forall E : \llbracket \text{entity}(E)\rrbracket^\zeta \land \llbracket-\text{entity}(E)\rrbracket^\zeta_0 \rightarrow E \in \text{Del}_r$

$b) \forall C : \llbracket \text{connection}(C)\rrbracket^\zeta \land \llbracket-\text{connection}(C)\rrbracket^\zeta_0 \rightarrow C \in \text{Del}_r$

c) $\forall M : \llbracket \text{mapping}(M)\rrbracket^\zeta \land \llbracket-\text{mapping}(M)\rrbracket^\zeta_0 \rightarrow M \in \text{Del}_r$

As an abbreviation we write $\forall Q : \llbracket-\text{element}(Q)\rrbracket^\zeta \land \llbracket-\text{element}(Q)\rrbracket^\zeta_0 \rightarrow Q \in \text{Del}_r$.

Note that since the removal of refinement and component-of relations may result in the deletion of further refinement relations, we cannot state for sure that exactly those model elements are removed that are prescribed by the rule. However, this is not in contradiction with the traditional DPO approach [45] as these side effects never occur for graph elements (i.e., entities, connections and mappings) only for refinement relations. This is the price we have to pay for our rich refinement hierarchy, which is more general than the typeless DPO approach or the traditional type graph formalization [44].

Adding new elements

When a rule prescribes the addition of new VPM elements, we build upon again the elementary VPM operations discussed in Sec. 3.4. We only discuss how to build upon the meta-level operations, but our approach can trivially handle the model-level operations as well in a similar way.

**Definition 4.23 (Extended matching).** Let $r = (\text{Pre}, \text{Post}, \text{par})$ be a graph transformation rule (with $Lhs$ as the root element of $\text{Pre}$) applied in a VPM model state $\mathfrak{A}$. An extended matching is a variable assignment $\zeta = \zeta_{\text{match,new}}$ where $\llbracket \varphi_{\text{Lhs}}(X_{\text{Lhs}})\rrbracket^\zeta = \text{true} \land \forall q \in \text{Post} \land q \not\in \text{Lhs} : Q = X_q \rightarrow \exists q. \text{id} : \{X_q \mapsto q. \text{id}\} \in \text{assign}$.

Informally, an extended matching (i.e., a matching pattern extended with variable assignment for new elements) is a variable assignment where a constant value is consistently assigned to each variable in $Lhs \cup Post$.

**Definition 4.24 (Addition of elements).** Let $r = (\text{Pre}, \text{Post}, \text{par})$ be a graph transformation rule (with $Lhs$ as the root element of $\text{Pre}$) applied in a VPM model state $\mathfrak{A}$ and extended matching $\zeta$.

The addition of elements prescribed by $\text{Post} \setminus \text{Lhs}$ is defined in Table 4.4.

<table>
<thead>
<tr>
<th>Add</th>
<th>VPM condition</th>
<th>Abbrev</th>
<th>ASM statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_e$</td>
<td>$e \not\in \text{Lhs} \land e \in \text{Post}$</td>
<td>add $e$</td>
<td>addEntity($e$)</td>
</tr>
<tr>
<td>$v_f$</td>
<td>$f(a, b) \not\in \text{Lhs} \land f(a, b) \in \text{Post}$</td>
<td>add $f$</td>
<td>addMappingFromAtom($F$, $A$, $B$)</td>
</tr>
<tr>
<td>$v_r$</td>
<td>$r(a, b) \not\in \text{Lhs} \land r(a, b) \in \text{Post}$</td>
<td>add $r$</td>
<td>addConnectionFromAtom($R$, $A$, $B$)</td>
</tr>
<tr>
<td>$v\sup$</td>
<td>$(a \rightarrow b) \not\in \text{Lhs} \land (a \rightarrow b) \in \text{Post}$</td>
<td>add $\sup$</td>
<td>addTwoToSuperType($A$, $B$)</td>
</tr>
<tr>
<td>$v\text{inst}$</td>
<td>$(A \rightarrow b) \not\in \text{Lhs} \land (A \rightarrow b) \in \text{Post}$</td>
<td>add $\text{inst}$</td>
<td>addInstanceOf($A$, $B$)</td>
</tr>
<tr>
<td>$v\text{comp}$</td>
<td>$(\exists k : a = b[k]) \not\in \text{Lhs} \land (\exists k : a = b[k]) \in \text{Post}$</td>
<td>add $\text{comp}$</td>
<td>addTwoToComponentOf($A$, $B$)</td>
</tr>
</tbody>
</table>

Table 4.4. ASM statements for additions by a rule

The corresponding ASM rule that carries out the required additions is the following:

**Rule** $\text{Add}_r(\overline{X}_{\text{add}}) = \text{seq add}_e; \text{seq add}_f; \text{seq add}_r; \text{seq add}_\sup; \text{seq add}_\text{inst}; \text{seq add}_\text{comp};$

where $\overline{X}_{\text{add}} = \langle X_{q_1}, \ldots X_{q_\ell} \rangle$ and all $q_i$ such that $q_i \in \text{Post} \land q_i \not\in \text{Lhs}$ are variables.

Let $\text{Add}_r$ denote the set of elements that are explicitly added by rule $r$, i.e., $\forall q : v_q \rightarrow Q \in \text{Add}_r$. 
**Proposition 4.25 (Correctness of addition).** Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as the root element of \( \text{Pre} \)) applied in a consistent VPM model state \( \mathcal{B} \) and an extended matching \( \zeta \). Then for all \( \mathcal{C} = \text{next}_{\text{add}}(\mathcal{B}, \zeta) \),

1. state \( \mathcal{C} \) is consistent
2. if all graph elements \( q \in \text{Post} \setminus \text{Lhs} \) (i.e., entities, connections and mappings) that are aimed to be created are non-existent then the corresponding new elements are successfully added. Formally,
   a) \( \forall e \notin \text{Lhs} \land e \in \text{Post} : [\text{-entity}(E)]_{\mathcal{B}}^{\mathcal{C}} \rightarrow [\text{-entity}(E)]_{\mathcal{C}}^{\mathcal{C}} \)
   b) \( \forall c \notin \text{Lhs} \land c \in \text{Post} : [\text{-connection}(C)]_{\mathcal{B}}^{\mathcal{C}} \rightarrow [\text{-connection}(C)]_{\mathcal{C}}^{\mathcal{C}} \)
   c) \( \forall m \notin \text{Lhs} \land c \in \text{Post} : [\text{-mapping}(M)]_{\mathcal{B}}^{\mathcal{C}} \rightarrow [\text{-mapping}(M)]_{\mathcal{C}}^{\mathcal{C}} \)

   As an abbreviation we write \( \forall q \notin \text{Lhs} \land q \in \text{Post} : [\text{-element}(Q)]_{\mathcal{B}}^{\mathcal{C}} \rightarrow [\text{-element}(Q)]_{\mathcal{C}}^{\mathcal{C}} \)
3. Nothing else but what prescribed by \( \text{Post} \setminus \text{Lhs} \) is created as a result of rule application. Formally,
   a) \( \forall E : [\text{-entity}(E)]_{\mathcal{B}}^{\mathcal{C}} \land [\text{-entity}(E)]_{\mathcal{C}}^{\mathcal{C}} \rightarrow E \in \text{Add}_E \)
   b) \( \forall C : [\text{-connection}(C)]_{\mathcal{B}}^{\mathcal{C}} \land [\text{-connection}(C)]_{\mathcal{C}}^{\mathcal{C}} \rightarrow C \in \text{Add}_C \)
   c) \( \forall M : [\text{-mapping}(M)]_{\mathcal{B}}^{\mathcal{C}} \land [\text{-mapping}(M)]_{\mathcal{C}}^{\mathcal{C}} \rightarrow M \in \text{Add}_M \)

   As an abbreviation we write \( \forall Q : [\text{-element}(Q)]_{\mathcal{B}}^{\mathcal{C}} \land [\text{-element}(Q)]_{\mathcal{C}}^{\mathcal{C}} \rightarrow Q \in \text{Add}_Q \).

If we assume that all elements in \( \text{Post} \setminus \text{Lhs} \) are variables (i.e., new elements do not have predefined identities) then by using the create ASM construct we can guarantee that fresh identities assigned to these variables are taken from the reserve. This, in turn, implies that the preconditions of elementary VPM operations for addition trivially hold.

**Corollary 4.26.** Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as the root element of \( \text{Pre} \)) applied in a consistent VPM model state \( \mathcal{B} \) and an extended matching \( \zeta \). If \( \forall q \notin \text{Post} \land q \notin \text{Lhs} : Q = X_q \) (i.e., all elements in \( \text{Post} \setminus \text{Lhs} \) are variables) then for all subsequent state \( \mathcal{C} = \text{next}_{\text{add}}(\mathcal{B}, \zeta) \), the corresponding VPM elements will become existent in this new state, formally,

1. \( \forall e \notin \text{Lhs} \land e \in \text{Post} : [\text{-entity}(E)]_{\mathcal{B}}^{\mathcal{C}} \)
2. \( \forall c \notin \text{Lhs} \land c \in \text{Post} : [\text{-connection}(C)]_{\mathcal{B}}^{\mathcal{C}} \)
3. \( \forall m \notin \text{Lhs} \land c \in \text{Post} : [\text{-mapping}(M)]_{\mathcal{B}}^{\mathcal{C}} \)

As an abbreviation we write \( \forall q \notin \text{Lhs} \land q \in \text{Post} : [\text{-element}(Q)]_{\mathcal{B}}^{\mathcal{C}} \)

Now we have all auxiliary constructs to define an ASM semantics for graph transformation.

**Application of a rule (ASM semantics)**

**Definition 4.27 (application of a rule on a matching).** Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as root element of \( \text{Pre} \)), \( \zeta \) an extended matching that contains an assignment at least for variables \( \overline{X}_{\text{Lhs}} \) and \( \overline{X}_{\text{add}} \).

The application of rule \( r \) to a VPM model state \( \mathcal{A} \) on matching assign (in other terms, a graph transformation step) rewrites the state \( \mathcal{A} \) into state \( \mathcal{B} \) (denoted as \( \mathcal{B} = \text{next}_{\text{r}}(\mathcal{A}, \zeta) \) in ASM terms) by the ASM rule defined in Algorithm 15.

Our main theorem of this section establishes that the correctness of rule application, if a rule is applicable (i.e., the LHS of the rule was successfully matched and negative application conditions are also satisfied), the proper elements are modified (created or removed), and the result is a well-formed VPM model (thus it is a graph according to Invariant 3.32).

**Theorem 4.28 (Correctness of a graph transformation step).** Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with \( \text{Lhs} \) as the root element of \( \text{Pre} \) and with only variables in \( \text{Post} \setminus \text{Lhs} \)),
Algorithm 15 ASM semantics for a rule application on a given matching (candidate)

\begin{verbatim}
rule ρ_{dpe}(\bar{X}_{ls}, \bar{X}_{add}) =
  if \phi_{ls}(\bar{X}_{ls}) \land \eta_r \land \delta_r then {with identification and dangling condition} 
    delete_{dpe}(\bar{X}_{del}); {with soft delete operations} 
    add_r(\bar{X}_{add}) 
  end if 

rule ρ_{spo}(\bar{X}_{ls}, \bar{X}_{add}) =
  if \phi_{ls}(\bar{X}_{ls}) then 
    delete_{spo}(\bar{X}_{del}); {with forced delete operations} 
    add_r(\bar{X}_{add}) 
  end if
\end{verbatim}

and $\mathfrak{A}$ a consistent VPM model state. Moreover, let $\zeta$ be an extended matching that contains an assignment at least for variables $\bar{X}_{ls}$ and $\bar{X}_{add}$.

Now if $[\phi_{ls}(\bar{X}_{ls}) \land \eta_r \land \delta_r]_\zeta = \text{true}$ (i.e., the identification and dangling conditions are fulfilled as well) then for all states $\mathcal{C} = \text{next}_{dpe}(\mathfrak{A}, \zeta)$ the following holds.

1. $\forall q \in Lhs \land q \notin Post : [\neg \text{element}(Q)]_\zeta$ (all elements mapped to a rule element in $Lhs \setminus Post$ are removed)
2. $\forall Q : [\text{element}(Q)]_\zeta^{3} \land [\neg \text{element}(Q)]_\zeta^{3} \rightarrow Q \in Del_r$ (nothing else is removed)
3. $\forall q \notin Lhs \land q \in Post : [\text{element}(Q)]_\zeta$ (all elements mapped to a rule element in $Post \setminus Lhs$ are created)
4. $\forall Q : [\neg \text{element}(Q)]_\zeta^{3} \land [\text{element}(Q)]_\zeta^{3} \rightarrow Q \in Add_r$ (nothing else is created).

Now if $[\phi_{ls}(\bar{X}_{ls})]_\zeta^{3} = \text{true}$ then for all states $\mathcal{C} = \text{next}_{spo}(\mathfrak{A}, \zeta)$ Statements 1, 3, and 4 hold (but Statement 2 does not necessary hold since dangling links are removed).

Note that if the matching is fixed then the result of rule application is unique up to an isomorphism. In case of individual graph transformation steps, we can simply create a variable assignment that becomes an extended matching as illustrated in Algorithm 16. However, this basic strategy will be extended later on in Sec. 4.4.1 to handle several rule application modes.

Definition 4.29. Let $r = (Pre, Post, par)$ be a graph transformation rule (with $Lhs$ as root element of $Pre$) with an ASM rule $\rho_{dpe}$ ($\rho_{spo}$) defined as semantics according to Def. 4.27.

A (single) application of rule $r$ to a VPM model state $\mathfrak{A}$ (in other terms, a non-deterministic graph transformation step) rewrites the state $\mathcal{A}_M$ into state $\mathfrak{B}$ (denoted as $\mathfrak{B} = \text{next}_r(\mathfrak{A})$) by the ASM rule defined in Algorithm 16.

Algorithm 16 ASM semantics for a single, non-deterministic rule application

\begin{verbatim}
rule once_{dpe} = 
  choose \bar{X}_{ls} with $[\phi_{ls}(\bar{X}_{ls}) \land \eta_r \land \delta_r]_\zeta = \text{true} do 
    create \bar{X}_{add} do 
      r_{dpe}(\bar{X}_{ls}, \bar{X}_{add}) 
    end create
  end choose

rule once_{spo} = 
  choose \bar{X}_{ls} with $[\phi_{ls}(\bar{X}_{ls})]_\zeta = \text{true} do 
    create \bar{X}_{add} do 
      r_{spo}(\bar{X}_{ls}, \bar{X}_{add}) 
    end create 
end choose
\end{verbatim}
4.3 An ASM Semantics for Graph Transformation

Prior to applying the previous propositions, we have to show that the \texttt{choose} and \texttt{create} constructs defined an extended matching.

**Proposition 4.30.** The variable assignment \( \zeta \) defined by the \texttt{choose} and \texttt{create} constructs is an extended matching.

**Example 4.31.** The meta-level ASM representation of the firing rule of finite automaton (stepFA depicted in Fig. 4.6) executed according to the SPO approach is as follows.

\[
\text{rule } \text{once(stepFA) = } \\
\text{choose } A1, S1, S2, T1, C1, C2, C3, C4, C5, C6 \text{ with } \\
\text{entity(automaton)} \land \text{entity(state)} \land \text{entity(transition)} \land \\
\text{connection(states)} \land \text{from(states)} = \text{automaton } \land \text{to(states)} = \text{state } \land \\
\text{connection(transitions)} \land \text{from(transitions)} = \text{automaton } \land \text{to(transitions)} = \text{transition } \land \\
\text{connection(from)} \land \text{from(from)} = \text{transition } \land \text{to(from)} = \text{state } \land \\
\text{connection(current)} \land \text{from(current)} = \text{automaton } \land \text{to(current)} = \text{state } \land \\
\text{entity(A1)} \land \text{entity(S1)} \land \text{entity(S2)} \land \text{entity(T1)} \land \\
\text{instanceOf(A1, automaton)} \land \text{instanceOf(S1, state)} \land \\
\text{instanceOf(S2, state)} \land \text{instanceOf(T1, transition)} \land \\
\text{connection(C1)instanceOf(C1, states)} \land \text{from(C1)} = \text{A1 } \land \text{to(C1)} = \text{S1 } \land \\
\text{connection(C2)instanceOf(C2, states)} \land \text{from(C2)} = \text{A1 } \land \text{to(C2)} = \text{S2 } \land \\
\text{connection(C3)instanceOf(C3, transitions)} \land \text{from(C3)} = \text{A1 } \land \text{to(C3)} = \text{T1 } \land \\
\text{connection(C4)instanceOf(C4, from)} \land \text{from(C4)} = \text{T1 } \land \text{to(C4)} = \text{S1 } \land \\
\text{connection(C5)instanceOf(C5, to)} \land \text{from(C5)} = \text{T1 } \land \text{to(C5)} = \text{S2 } \land \\
\text{connection(C6)instanceOf(C6, current)} \land \text{from(C6)} = \text{A1 } \land \text{to(C6)} = \text{S1 } \text{ do } \\
\text{create } C7 \text{ do } \\
\text{delConnectionRfromAtoB(C6, A1, S1) } \\
\text{addConnectionRfromAtoB(C7, A1, S2) } \\
\text{xGenerateOfY(C7, current) } \\
\text{end create } \\
\text{end choose}
\]

**Example 4.32.** The model-level ASM representation of the firing rule of the finite automaton (stepFA depicted in Fig. 4.6) executed according to the SPO approach is as follows.

\[
\text{rule } \text{once(stepFA) = } \\
\text{choose } A1, S1, S2, T1, C1, C2, C3, C4, C5, C6 \text{ with } \\
\text{automaton(A1)} \land \text{state(S1)} \land \text{state(S2)} \land \text{state(C1, A1, S1)} \land \text{state(C2, A1, S2)} \land \\
\text{transition(T1)} \land \text{transitions(C3, A1, T1)} \land \text{from(C4, T1, S1)} \land \text{to(C5, T1, S2)} \land \text{current(C6, A1, S1)} \text{ do } \\
\text{create } C7 \text{ do } \\
\text{delConnectionCurrent(C6, A1, S1) } \\
\text{addConnectionCurrent(C7, A1, S2) } \\
\text{end create } \\
\text{end choose}
\]

As a result of this section, we formalized the behavior traditionalDPO and SPO approaches of graph transformation to carry out high-level rule based manipulation of VPM models. More specifically, we described how to apply a rule once on a matching as abstract state machines.

### 4.3.5 Transformation sequences and concurrency

The semantics of a modeling language is typically not defined by a single rule application but rather a sequence of consecutive transformation steps.
Definition 4.33 (Transformation sequence). A transformation sequence $M_0 \xrightarrow{r_1(o_1)} \ldots \xrightarrow{r_n(o_n)} M_n$ is a sequence of consecutive transformation steps using the rules $\{r_0, \ldots, r_n\}$.

Graph transformation can be non-deterministic in two different ways: the user has a choice

1. when selecting the next rule to be applied and
2. when selecting the matching on which the previously selected rule is applied on.

This non-deterministic nature makes graph transformation a very powerful modeling paradigm as the transformation designer can easily avoid the overspecification of behavior.

However, for many practical cases, we have to guarantee that the results of graph transformation are unique (or confluent) in the sense that a unique model is obtained after applying arbitrarily chosen rules as long as possible (and possibly in parallel) on a given initial model. To be precise this uniqueness is interpreted up to an isomorphism, i.e., we consider two models differing from each other only in object identifiers to be identical. For the concurrency results and definitions, we basically follow [77, 142].

The theory of graph transformation offers valuable results on handling such concurrency in order to guarantee that the results of transformation sequences are unique (see [142] for further details on Local Church-Rosser Theorem and the Parallelism Theorem).

Sequential independence of transformation steps highlights situations when a rule application is not causally dependent on the consecutive transformation step. In this sense, a transformation engine may swap such transformation steps, for instance, for optimizing performance.

Definition 4.34 (Sequential independence vs. causal dependence). Two consecutive transformation steps $M_0 \xrightarrow{r_1(o_1)} M_1 \xrightarrow{r_2(o_2)} M_2$ are sequentially independent if the occurrence (matching) $o_1$ of the Rhs of $r_1$ and the matching $o_2$ of the Lhs pattern of $r_2$ only overlap in elements that are preserved by both steps. Otherwise, we say that the two steps are causally dependent.

In other terms, to achieve sequential independence, the first transformation step must not create model elements that are required for matching the positive patterns in the second transformation step, moreover, the second transformation step must not remove model elements that were successfully matched by positive patterns and then preserved in the first transformation step.

Example 4.35 (causally dependent steps). Two consecutive transformation steps applied on our running finite automaton example (see Fig. 2.3 for the model, and Fig. 4.6 for the transformation rule) are depicted in Fig. 4.8, which turn out to be causally dependent on each other.

In order to judge sequential independence, we have to compare the second row in the figure, which highlights the occurrence of the Rhs for the first transformation step (Step 1 on the left) and the occurrence of the Lhs for the second transformation step (Step 2 on the right).

We notice that there are overlapping elements, namely automaton $a_1$, state $s_2$, and the states and current links between them (we used abbreviation for clarity reasons for link names). However, while the first three elements are preserved by both steps (i.e., the structure of the finite automaton is not modified), the current link is not preserved, in fact, by any of the steps. Therefore, we can conclude that the two steps are causally dependent.

It is worth noted for a comparison that this definition of independence is more sophisticated than the traditional notion of independent firings of transitions in Petri nets, as it allows to use shared resources that are not modified.

For performance reasons by reducing intermediate states, a transformation engine is frequently necessitated to group individual transformation steps into a complex "macro" step and execute them in parallel. However, in order to attain a consistent result, only such steps can be grouped together that are not in a conflict. Informally, a conflict occurs when the two transformation steps would execute
contradicting operations on the same model element (i.e., one of them removes it while the other preserves it).

**Definition 4.36 (parallel independence vs. conflicts).** Two transformation steps \( M_0 \xrightarrow{r_1(a_1)} M_1 \) and \( M_0 \xrightarrow{r_2(a_2)} M_2 \) are **parallely independent** if the occurrence (matchings) of the Lhs pattern of \( r_1 \) and the occurrence of the Lhs pattern of \( r_2 \) only overlap in elements that are preserved by both steps. Otherwise, we say that the two steps are in **conflict**.

Informally speaking, in case of parallely applied rules, one rule application must not remove elements that are required for the successful matching of the positive patterns at the other rule application, and it must not create new elements that are prohibited by the negative patterns of the other rule application.

**Example 4.37 (conflicting transformation steps).** Two parallel transformation steps on our finite automaton example are depicted in Fig. 4.9, which are now proved to be conflicting.

In order to judge parallel independence, we have to compare the first row in the figure, which highlights the occurrence of the Lhs for one transformation step (Step 1a on the left) and the occurrence of the Lhs for the other transformation step (Step 1b on the right).

We notice that there are overlapping elements, namely automaton \( s1 \), state \( s1 \), and the states and current links between them. However, while the first three elements are preserved by both steps, the current link is not preserved. Therefore, we can conclude that the two steps are conflicting.

Note that definitions speak about conflict and causal dependency of rule applications and not the conflict and causal dependency of rules themselves. However, transformation designers may detect
potentially conflicting situations for rules by the sophisticated static analysis techniques of critical pair analysis [36, 85] (implemented in the AGG tool [64]). In this respect, we can sometimes guarantee that a graph transformation system (as it is, i.e., for all transformation sequences) is free of conflicts or causal dependencies, for instance.

While the causal dependency of rule applications can be a natural characteristic of a transformation (a rule is only applicable after the successful application of another), conflicts are frequently much more problematic (especially, for model transformations between modeling languages).

A possibility to eliminate conflicts is to introduce additional positive and negative conditions to patterns of rules, which result in new causal dependencies between them. However, following this approach, we would easily end up with graph transformation rules with very complex patterns. As an alternate solution, explicit control structures can be introduced to restrict which rule (or rules) is potentially applicable at a time.

4.4 Visual Definition of Modeling Languages and Model Transformations

In the current section we define additional control structures to restrict the allowed transformation sequences of graph transformation. For this reason, we define the concepts of model transition systems, which provides a uniform description mechanism for both modeling languages and model transformations.

4.4.1 Control structures

Typically, we cannot define the operational semantics of a modeling language with a single graph transformation rule, but a set of interacting rules is required.

Definition 4.38 (Graph transformation system). A graph transformation system \( GTS \) is a pair \((MM, R)\) that consists of a metamodel \( MM \) and a set \( R \) of graph transformation rules over \( MM \).

Definition 4.39 (Graph grammar). A graph grammar \( GG = (M_0, GTS) \) consists of an initial (instance) model \( M_0 \) and a graph transformation system \( GTS \).
However, in many practical cases, the sequence of rule applications must be restricted in addition, thus we introduce control structures. In order to motivate the use of control structures, we formalize the reachability problem of finite automata with control structures in Fig. 4.10. According to our informal specification, all initial states of an automaton are reachable by definition. Then all states of the finite automaton that can be reached from the initial state by firing a certain transition sequence are reachable as well. The control structures will impose this transitive nature of the reachability problem on rule applications.

Example 4.40 (Reachability problem for finite automata). Rule initR states that all states of the automaton marked as initial are reachable (if the state has not been marked previously). When applying this rule to the finite automaton model of Fig. 2.3, a new reachable link is created leading from s1 to s1.

Rule reachR expresses that if a reachable state s1 of the automaton is connected by a transition T1 to such a state s2 that is not reachable yet then s2 should also become reachable as a result of the rule application.

The control structure prescribes that initR should be executed parallely for all matchings (in forall mode) prior to applying reachR as long as possible (in loop mode).

Note that without the negative application condition, the transformation would generate more than a single reachable link between an automaton and a state, which would contradict our requirements.

Definition 4.41 (Control flow graph). A control flow graph CFG is a graph with the following node and edge types.

- There are the following types of nodes of a CFG: Start, End, Nondet, Try, Forall, Loop and Call.
- There are two types of edges: succeed and fail.

The control flow graph is evaluated by a virtual machine which traverses this graph according to the edges and applies the rules associated to each node.

1. The execution starts in the Start and finishes in the End node. Neither types of nodes have rules associated to them.
2. When a Nondet node is reached, one of the enabled rules associated to the node which are enabled is selected non-deterministically for execution. If there was at least one enabled rule then the next node is determined by the succeed edge, otherwise the fail edge is followed.
3. When a **Try** node is reached, its associated rule is tried to be executed. If the rule was applied successfully, then the next node is determined by the **succeed** edge, while in case the execution failed, the **fail** edge is followed.

4. At a **Loop** node, the associated rule is applied as long as possible (which may cause non-termination in the macro step).

5. When a **ForAll** node is reached, the related rule is executed parallelly for all distinct (possible none) occurrences in the current host graph.

6. At a **Call** node (which has an associated CFG and not a rule) the state of the CFG machine is saved and the execution of the associated CFG is started (in analogy with function calls in programming languages). When the sub CFG machine is terminated, the saved state is restored, and the execution is continued in accordance with the outgoing edge (**succeed** or **fail**).

Note that these control structures do not overspecify control, since by looping a **Nondet** node that contains all the rules of a GTS, we obtain the traditional (uncontrolled) semantics of graph transformation. However, nodes in the CFG can be interpreted as high-level “macro” steps for the entire transformation process composed of elementary rule applications.

From a theoretical point of view, note that these control structures are extensions of the ones defined in [78], where the authors prove that they provide a minimal set of functionally complete control structures.

The formal semantics of these rule application modes is defined again by abstract state machines. For this purpose, we typically need to check the precondition of the rule prior to the rule application itself in order to determine the next rule to be applied (see try and call rule applications).

**Definition 4.42 (ASM semantics of control structures).** The ASM semantics of elementary control structures is defined in Algorithm 17 (only for the SPO approach since control structures are approach independent).

### 4.4.2 Defining modeling languages: dynamic behavior

Now we define the concepts of model transition systems (with slight technical modifications to the definitions in [167,168]) that serves the formal framework for defining operational semantics for visual modeling languages and their model instances.

**Definition 4.43 (Modeling language).** A visual **modeling language** \(ML = (GTS, CFG)\) is a pair, where \(GTS\) is a graph transformation system (i.e., a set of graph transformation rules typed over a metamodel) and \(CFG\) is a set of a **control flow graphs** (CFG).

Now we can define the operational semantics of a model as a special graph grammar.

**Definition 4.44 (Dynamic operational semantics of a model).** A **model transition system** \(MTS = (Init, ML)\) is a pair that defines the dynamic operational semantics of the **initial model** \(Init\), which is a well-formed instance of the metamodel of modeling language \(ML\).

As a final and more practical remark, a good modeling practice that should be followed by transformation designers is as follows.

**Proposal 4.45 (Separation of phases in language definition).** In all practical cases, the control structure of a language should very clearly be separated into a terminating **initialization phase** (aiming at to derived auxiliary elements) and a potentially non-terminating **execution phase** (specifying the dynamic behavior). Naturally, in case of simple modeling languages, the initial phase is often superfluous.

To summarize our concepts, a precise definition of a modeling language should always contain its metamodel, a set of graph transformation rules possibly constrained by a control flow graph.
Algorithm 17 ASM semantics for control structures

rule nondet\{r_1, \ldots , r_n\} =
choose r with r \in \{r_1, \ldots , r_n\} \land \exists \overline{X}_{Lhs} : [\phi_{r,Lhs}(\overline{X}_{Lhs})]_\delta^\alpha = \text{true} do
choose \overline{X}_{Lhs} with [\phi_{r,Lhs}(\overline{X}_{Lhs})]_\delta^\alpha = \text{true} do
create \overline{X}_{add} do r(\overline{X}_{Lhs}, \overline{X}_{add})
end choose
end choose

rule try\{r\} : \text{bool} =
if \exists \overline{X}_{Lhs} with [\phi_{r,Lhs}(\overline{X}_{Lhs})]_\delta^\alpha = \text{true} then
choose \overline{X}_{Lhs} with [\phi_{r,Lhs}(\overline{X}_{Lhs})]_\delta^\alpha = \text{true} do
create \overline{X}_{add} do r(\overline{X}_{Lhs}, \overline{X}_{add})
end choose
return \text{true}
else
return \text{false}
end if

rule loop\{r\} =
while \exists \overline{X}_{Lhs} with [\phi_{r,Lhs}(\overline{X}_{Lhs})]_\delta^\alpha = \text{true} do
choose \overline{X}_{Lhs} with [\phi_{r,Lhs}(\overline{X}_{Lhs})]_\delta^\alpha = \text{true} do
create \overline{X}_{add} do r(\overline{X}_{Lhs}, \overline{X}_{add})
end choose
end while

rule forall\{r\} =
forall \overline{X}_{Lhs} with [\phi_{r,Lhs}(\overline{X}_{Lhs})]_\delta^\alpha = \text{true} do
create \overline{X}_{add} do r(\overline{X}_{Lhs}, \overline{X}_{add})
end choose

rule call\{cfg\} =
let \text{r} = \text{cfg.start in r}

4.4.3 Static semantics of a language

Metamodels are typically insufficient to define all the additional well-formedness constraints (static semantics) for model instances. For instance, we cannot express in the metamodel of finite automata that, for instance, no transitions are allowed to lead into an initial state supposing that this is the user’s intention.

Therefore, a constraint language is required to express such restrictions on well-formed model instances. The standard way of expressing such constraints is the use of the Object Constraint Language (OCL), which is an object-oriented textual language part of the UML standard. OCL constraints are side-effect free, i.e., they cannot modify the state of the system. However, they can prescribe invariants (logic properties that must hold in each state of the system) or pre- and post-conditions for operations.

On the other hand, it is worth pointing out (as done in [169]), that graph patterns (or graph transformation rules with identical LHS and RHS) may also serve as a visual constraint language.

Moreover, since a metamodel of OCL has been already constructed (first in [141]), standard OCL expressions can be easily transformed into graph patterns by using our model transformation framework. In this view, both the transformation process (from OCL to graph patterns) both the validation process (when examining graph patterns) could be performed within a single framework.

As in the current thesis, we are focusing on dynamic behavior of models, all questions of static well-formedness are out of the scope (although we believe that many parts of our framework would directly be applicable). Therefore, for the rest of the thesis, we assume that models to be investigated fulfill all static well-formedness constraints specified somehow in an arbitrary constraint language.
4.4.4 Model transformation systems

Model transition systems can easily be tailored to the special needs of model transformations in order to provide a precise (but still practice oriented) mathematical background.

Reference models and metamodels

As the main goal of model transformation is to derive a target model from a given source model, source and target objects must be linked to each other in some way to form a single model. For this reason, the following definition introduces the concepts of a reference model (graph). The structure of a reference model is also constrained by a corresponding metamodel, which contains (i) references of existing source and target metamodel objects; (ii) new (so-called) reference objects that provide a typed coupling of source and target objects, and (iii) reference links connecting all these objects.

Definition 4.46 (Reference metamodel).

A reference metamodel \( MM_{ref} = (MM_{src}, MM_{trg}, CLS_{ref}, ASSOC_{ref}, ATTR_{ref}) \) imports (or contains) a source and a target metamodel \( (MM_{src}, MM_{trg} \) respectively), and an additional set of reference classes \( CLS_{ref} \), associations \( ASSOC_{ref} \), and attributes \( ATTR_{ref} \), where reference associations may lead from a reference class to either a source, a target or a reference element of a specific class.

Definition 4.47 (Reference model).

A reference model \( M_{ref} = (M_{src}, M_{trg}, OBJ_{ref}, LNK_{ref}, SLT_{ref}) \) contains a source and a target model \( (M_{src}, M_{trg} \) respectively) that corresponds to their own metamodels, and an additional set of reference objects \( NODES_{ref} \), links \( LNK_{ref} \), and slots \( SLT_{ref} \), which correspond to the reference metamodel.

Naturally, reference metamodels and models can be uniformly represented as VPM models.

Definition 4.48 (VPM representation of a reference metamodel/model).

Reference metamodels (and models) are well-formed VPM entities \( E_{MM} \) where the entities representing the source and target language \( (E_{src}, E_{trg}, \) respectively) are explicitly contained by \( E_{MM} \), i.e., \( \exists i, j : E_{src} = E_{MM}^{(i)} \land E_{trg} = E_{MM}^{(j)} \).

Reference models preserve the mapping information between corresponding source and target models. In this sense, they provide a primary basis for back-annotation of analysis results as long as the results of the mathematical analysis are of structural nature (for instance, they highlight a component in the system that is erroneous or not reliable enough). However, in case of traces for counter examples yielded by model checker tools, there is an implicit model transformation within the analysis tool, therefore reference models serve as milestones for back-annotation but do not completely solve the problem.

Proposal 4.49 (Back-annotation on reference models). Back-annotation of analysis results can be implemented on the basis of reference graphs supposing that input formalism of the analysis tool is identical to its formalism for the results.

Model transformation rules

The adaption of graph transformation rules to model transformations prescribe special requirements for the structure of these rules. As the target model is constructed from scratch, model transformation rules are frequently non-deleting, which ensures the pleasant property of being able to handle all the LHS matches parallelly.
On the other hand, when the deletion of certain graph objects is prescribed by a rule, we must ensure that distinct parallel matches do not conflict with each other. In our model transformation approach, parallelly executable rules cannot remove any part of the graph to avoid such problems.

Following the classification of different graph transformation approaches that can be found in [142], a model transformation rule is defined as follows.

**Definition 4.50 (Model transformation rule).** A model transformation rule \( r_{mt} \) is a special graph transformation rule, where

- all graphs \( Lhs, Neg, \) and \( Rhs \) are reference graphs;
- an occurrence of \( Lhs \) in \( M_{ref} \) is required to be an isomorphic image of \( Lhs \);
- the source model is left unaltered;
- all the dangling edges are deleted automatically.

As industrial applications of model transformation surely consist of very large and complex models containing several tens of rules, the flow of model transformation must be restricted this time as well by the same control mechanism as before that allows to construct transformations in a hierarchical way.

**Definition 4.51 (Model transformation system).** A model transformation system is a special modeling language \( ML_{mt} = (GTS_{mt}, CFG_{mt}) \) with a reference metamodel \( M_{ref} \) and model transformation rules \( r_{mt} \in R_{mt} \) in its graph transformation system \( GTS_{mt} = (M_{ref}, R_{mt}) \) component.

**Definition 4.52 (Model transformation).** A model transformation (instance/process) is a special model transition system \( MTS_{mt} = (M_{src}, ML_{mt}) \) with a model transformation system \( ML_{mt} \) as the modeling language and a model instance of the source language \( M_{src} \) as the initial model.

In this sense, we have a single specification formalism (by introducing an ASM semantics of graph transformation on VPM models) that is simultaneously applicable for modeling languages and their transformations, however, the typical structure of rules can be very different in the two cases.

- Rules for capturing operational semantics typically modify very small parts of a model (i.e., there are much fewer dynamic elements than static ones in a metamodel). On the other hand, both creation and deletion of model elements are common in such rules.
- Rules for specifying model transformations typically only create new model elements but do not remove them since they have to build up the reference and target models from scratch. As such a generation process includes the generation of static parts as well, the entire reference and target metamodel is considered to be dynamic.

Finally, we define a refinement of graph transformation rules in order to enable the reuse of dynamic behavior of modeling languages and model transformations.

### 4.5 Rule Refinement

A main goal of multilevel metamodeling is to allow a hierarchical and modular design of domain models and metamodels where the information gained from a specific domain can be reused (or extended) in future applications. Up to now, metamodeling approaches only dealt with the reuse of the static structure while the reuse of dynamic aspects has not been considered. However, it is a natural requirement (also in mathematical domains) that if a domain is modeled as a graph then all the operations defined in a library of graphs (such as node/edge addition/deletion, shortest path algorithms, depth first search, etc.) should be adatable for this specific domain without further modifications.
Moreover, semantic operations are frequently needed to be organized in a hierarchy (and executed accordingly). For instance, in feature/service modeling, we would like to express for the user that an operation copying highlighted text from one document to another and another operation that copying a selected file between two directories are conceptually similar in behavior when regarding from a proper level of abstraction. Thus a domain engineer should be able to derive the “text copying” operation from the abstract “copy a thing to a certain place” operations. In other terms, in analogy with traditional structural ontologies, dynamic behavior can also be classified into a hierarchy and reused in specific application domains.

To capture such semantic abstractions, we define rule refinement as a precise extension of metamodeling for dynamic aspects of a domain as follows, which is conceptually similar to the use of late bindings and method overriding in object-oriented programming languages.

Based upon the definition of pattern refinement (see Def. 2.18), the refinement relation of typed rules is defined as follows:

**Definition 4.53 (Rule refinement).** A rule $r_{sub} = (Lhs_{sub}, Rhs_{sub}, Neg_{sub})$ is a refinement of rule $r_{super} = (Lhs_{super}, Rhs_{super}, Neg_{super})$, denoted as $r_{sub} \subseteq r_{super}$, if

1. $Lhs_{sub} \subseteq Lhs_{super}$; the positive preconditions of $r_{super}$ are not stronger (more general) than of $r_{sub}$;
2. $Neg_{sub} \subseteq Neg_{super}$; the negative preconditions of $r_{super}$ are not stronger than of $Neg_{sub}$;
3. $Lhs_{sub} \cap Rhs_{sub} \subseteq Lhs_{super} \cap Rhs_{super}$: the preserved elements of $r_{super}$ are not stronger than of $r_{sub}$;
4. $Lhs_{sub} \setminus Rhs_{sub} \subseteq Lhs_{super} \setminus Rhs_{super}$: thus $r_{sub}$ removes at least the elements that are deleted by the application of $r_{super}$;
5. $Rhs_{sub} \setminus Lhs_{sub} \subseteq Rhs_{super} \setminus Lhs_{super}$: thus $r_{sub}$ adds at least the elements that are added by the application of $r_{super}$.

**Example 4.54 (Rule refinement).** The concepts of rule refinement are demonstrated on a brief example (see Fig. 4.11). Let us suppose that a garbage collector removes a $Node$ from the model space (by applying rule $delNodeR$) if the reference counter of the node (which collects the number of edges leading into the node) has been decremented to 0 (denoted by the attribute condition $ref = 0$). Thus a transformation sequence for garbage collection may only remove isolated nodes from the model space.

![Fig. 4.11. Rule refinement](image)

Meanwhile, in case of Petri nets, we may forbid the presence of tokens not assigned to a place. Therefore, even when the reference counter of a $Place$ (which used to be a refinement of $Node$) reaches 0, an additional test is required for checking the non-existence of tokens attached. If none of such tokens are found then the place $P$ can be safely removed (cf. rule $delPlaceR$).

**Proposition 4.55.** Rule $delPlaceR$ is a refinement of rule $delNodeR$. Therefore, in typical applications (visual model editors, etc.) $delPlaceR$ takes precedence of $delNodeR$.

**Proof.** The three steps of the proof are the following:
1. \( Lhs_{delPlaceR} \subseteq Lhs_{delNodeR} \) since \( Place \) (the type of \( P \)) is a refinement of \( Node \) (the type of \( N \)).
2. \( Lhs_{delPlaceR} \subseteq \neg \text{NodeR} \) since rule \( delNodeR \) has no negative conditions.
3. Conditions 3-5 trivially hold due to the empty right-hand sides of rules.

We provide a more complex example of rule refinement later in Sec. 8.1.4.

### 4.6 Conclusions

Based on the paradigm of graph transformation, I elaborated a mathematically precise and visual formalism that simultaneously supports the (i) meta-level definition of an operational semantics to an arbitrary modeling language and (ii) the high-level specification of model transformations within and between such modeling languages.

- **An ASM semantics of graph transformation.** I proposed a formal operational semantics to graph transformation rules (Sec. 4.3: following [172,184]) based on abstract state machines which captures the semantic differences of major graph transformation approaches.

- **Control structures.** In order to restrict the non-determinism of graph transformation, I defined basic control structures in the form of a control flow graph (Sec. 4.4.1 based on [49,91,165,167,175,178,181,182,186,191]).

- **Support for back-annotation.** In order to provide means for the back-annotation of the results of a formal analysis carried out on the system model, I introduced the concepts of reference models and metamodels (Sec. 4.4.4 based on [49,165,175,176,178,182,186-188,190,191]) that interrelate the elements of the source and target modeling language.

- **Refinement of dynamic behavior.** I proposed a refinement of graph transformation rules (Sec. 4.5 following [181,186]) in order to enable the reuse of dynamic behavior when specifying the operational semantics of further modeling languages and model transformations.

**Conceptual relevance**

A main conceptual novelty of the chapter is the ASM definition of various graph transformation approaches. In this respect, semantic differences between these approaches can be pinpointed mathematically precisely yet rather intuitively in contrast to the traditional category-theoretical foundations.

The introduction of control structures introduces a macro step semantics for modeling languages above the graph transformation micro steps. In this respect, we can first split the dynamic operational semantics of a complex modeling language into elementary transformations (micro steps) and then the entire semantics of the language is constructed using control structures. This approach is demonstrated for Petri nets and UML statecharts in Chapter 8.

A further important result is the refinement of graph transformation rules which allows to construct metamodel libraries by reusing dynamic behavior as well. As an analogy, we may also use the term “dynamic design patterns”. Different semantic variations of a modeling language can be easily derived from each other by using this technique. A more complex example on Petri nets can be found in Sec. 8.1.4.

Model transformations captured by graph transformation rules provide precise means to specify various kinds of abstractions frequently necessitated by back-end verification techniques aiming to carry out the analysis of UML models. In this respect, model transformation may serve as an abstraction tool for transformation designers. First the analysis-specific parameters are attached to the UML models on the metamodel-level (model enrichment) and then the irrelevant aspects are filtered by abstraction (such as unused diagrams, or abstractions from infinite to finite domains).
**Practical relevance**

The main practical benefit of the chapter is that transformation designers may specify a large variety of model transformations using their well-known UML environment either between two MDA models, or from an MDA model to a mathematical domain (as in HIDE [24]).

If a graph transformation interpreter is implemented above a metamodeling framework such as VPM, we obtain a meta-simulator, i.e., a simulator that can be used as a simulator for various modeling languages provided that the operational semantics of the modeling language is captured by model transformation systems. This allow an interactive user-guided validation of the semantics of languages and transformations in an early phase of the design. Moreover, since operations and data can be stored uniformly in VPM, such a meta-simulator would implement the Neumann principle for models. In fact, the ASM formalization of model transformation systems would directly yield such a meta-simulator for the .Net framework using the executable AsmL language.

Furthermore, the automated program generation techniques of Chapter 5 will enable to compile graph transformation rules into various implementation platforms and thus integrate formal specifications into industrial UML CASE tools.

The back-annotation of analysis results can be implemented using reference metamodels and models. Typically, this back-annotation requires to highlight certain parts in the source UML model (e.g., a dependability bottleneck) or to visualize an error trace (using UML sequence diagrams). For this purpose, the interconnections of the mathematical model transformations has to be preserved and explicitly stored during the entire analysis cycle. Naturally, reference metamodels (and models) are also ordinary models, thus they can be stored as traditional VPM models.

**VIATRA: Tool support for model transformations**

As a main practical result of the chapter, one can easily build model transformation tools using the very high-level visual mathematical formalism of graph transformation with special control structures. In fact, the VIATRA system (that we already discussed in Sec. 1.3–1.4) is exactly such a tool that aims to perform transformations between modeling languages defined by their MOF metamodels.

Models, metamodels, transformation rules and control structures are all specified in a UML notation by using a commercial UML CASE tool. Then the tool automatically generates a Prolog program implementing the transformation (to be discussed in details in Chapter 5) and then automatically generates the target equivalent of the source model in an XMI format. The reference information is also stored during the transformation to support back-annotation.

In the future, we aim to extend VIATRA with dynamic refinement facilities which would allow the reuse of partial transformations (as algorithms) in a totally different domain.

**Feasibility studies**

In Chapter 8, I demonstrate the practical feasibility of the approach by formalizing industrial strength benchmark modeling languages, namely, UML statecharts and Petri nets. Moreover, in Sec. 8.3 a model transformation is presented from statecharts to Petri nets aiming to provide access to Petri net-based analysis techniques in a UML environment.

All these case studies (and many others implemented by various master's and PhD students) have been part of a Hungarian research project [91] and they were tested on large UML models obtained from industrial partners to demonstrate the practical feasibility of our approach. According to these experiments, we may draw the conclusion that the larger the transformation is, the better support we can offer since the usability and legibility of our visual formalism versus an XSLT-based transformation transformation script (such as, for instance, [55]) is in analogy with using a structured programming language instead of assembly.

However, for space considerations, the source UML models used for demonstration purposes in the current thesis are kept relatively small.
Limitations

Concerning the back-annotation, the practical results we achieved were somewhat limited mainly due to technological problems. Basically, we can identify several technological gaps in the UML standard that hinders this back-annotation process:

1. The XMI standard [12] is not uniformly implemented by different tool vendors, therefore reusing a UML model in the UML tool of another vendor frequently fails.
2. The standard XML-based exchange format for UML diagrams have been introduced very recently for UML 2.0. As a result, different back-annotation programs have to be implemented for different UML CASE tools (unlike the general framework for back-annotation of the abstract syntax).
3. An API for manipulating the UML models of different tool vendors in a uniform way is still missing. As a result, only tool specific manipulations are available (such as the RoseScript language for Rational Rose) to trigger graphical or structural changes in the model.
4. The concept of a trace diagram is also missing from UML. The problem here lies in the fact that UML sequence diagrams cannot faithfully represent a counterexample retrieved by a model checker as the changes of attribute at an objects are not visible.

On the other hand, it is a conceptual problem, when the analysis itself is a model transformation, i.e., when the input and output of the analysis tool differs a lot from each other. For instance, a model checker obtains a transition system as input, and might retrieve a counter example as output, where we only have partial information on the interrelation of the analysis results and the source UML model.

As a summary, a general, UML tool independent solution for the back-annotation problem is infeasible due to the preliminary state of the UML standard (and tools).

In the upcoming chapter, the attention is turned towards providing automation of model transformations. More specifically, we investigate (i) a reflective model transformation based program generation approach (yielding a Prolog program as the output) and (ii) the use of standard Action Semantics expressions as transformation programs.
Automated Model and Program Generation for Transformations

I present automated means for generating transformation programs as implementations of model transformation systems in the form of Prolog programs and standard Action Semantics descriptions. As a result, a higher quality of model transformations is obtained in contrast to hand-coded and highly error-prone transformation scripts.

5.1 Automated Implementation of Model Transformations: An Overview

In the previous chapters, we demonstrated that the combination of metamodeling and graph transformation provides a visual and easy-to-understand way to specify model transformations (within and between modeling languages), which fits well to best engineering practice.

However, even if the formal specification of a transformation is mathematically precise (and probably formally verified), its implementation is still highly error prone due to a huge abstraction gap between visual UML models and formal mathematical descriptions. For this reason, the automated generation of a program that implements the transformation is also a major requirement for model transformations. For this reason, we need to investigate how the process of graph transformation can be implemented automatically and efficiently.

As implementing a graph transformation engine can be complicated due to the complex pattern matching phase (see later in Sec. 5.1.2), first we make an attempt to build the implementation on an existing standard with wide tool support.

5.1.1 XSLT as model transformation engine

Since MOF based models are frequently stored in the corresponding XMI representation (as done in VIATRA [49] as well), the XSLT (XSL Transformations) [198] standard is a primary candidate for implementing model transformations (see [55,71,136] for existing approaches), since it was designed to carry out transformations between XML documents having a strict tree structure. Moreover, a number of XSLT engines are readily available, thus allowing one to focus on describing the model transformations without implementing the transformation engine.

XSLT is based on the concept of matching parts of the source document based on its structure and associating this with the construction of the target document. In this sense, it is closely related to a class of graph transformation systems since (i) the matching process requires to navigate from one node to another in the source XML document using regular XPath [199] expressions (ii) and then we perform some additions on the target XML document. In this sense, an XSLT transformation process can be expressed with a graph transformation system without deleting rules (and thus necessarily with special negative application conditions to guarantee termination).

Unfortunately, the XSLT approach has several deficiencies both from a specification and implementation perspective.
• Specification problems
  - **Lack of subtyping in pattern matching.** The pattern matching of XSLT requires exact
types, i.e., subtyping is not allowed in the matching process, which immediately blows up the
size of the specification.
  - **XSLT is verbose.** Unfortunately, several experiments (e.g., [71]) demonstrate that the manual
design of XSLT transformation scripts does not scale up for complex model transformations as
even simple transformations require several thousand lines of code. Moreover, it is very difficult
to separate the source and target parts of the rules, and we also have to be aware of the syntactic
representation of the model not just the abstract structure (i.e., the metamodel).
  - **Transformations within a language.** The XSLT method is sufficient if we perform model
transformations between modeling languages as the source language is not modified in such a
case. But since XSLT is not allowed to modify the document it is processing, transformations
within the same model (e.g., to express the semantics of a modeling language) become very
complicated and inefficient. Essentially, the result of each transformation step has to be expressed
as a separate XML document where only a very small part of the model is modified while the
rest of the document is simply copied over and over again.

• Implementation problems
  - **XSLT is inefficient.** Since everything is passed by-value in XSLT, it results in a great deal
of structure copying. This problem becomes extremely severe, if the source XML document
contains many cross-references (links by identifiers), which means that the document is no
longer a tree but a graph. Unfortunately, navigation primitives of XSLT and XPath are unable
to properly handle such situations when the matching process should follow cross references
(which are dominant in all MOF based XMI models).
  - **Debugging.** Finally, XSLT engines do not provide any facilities to debug XSLT transformation
scripts.

We may conclude that XSLT is inappropriate both for specifying and implementing non-trivial
model transformations. Thus we will now investigate how existing tools implement a graph transfor-
mation step.

5.1.2 The complexity of graph transformation

The implementation of a graph transformation step is typically divided into the **graph pattern
matching** and the **modification** phases.

**Graph pattern matching**

During the pattern matching phase, we try to establish a mapping from a **pattern graph** \( G_p \) to a **model
(host) graph** \( G_m \). This problem is equivalent with the well-known **subgraph isomorphism problem,**
which is known to be NP-complete. To be more precise, its complexity is \( O(|N_m|^{|N_p|}) \) where \( |N_m| \) is
the number of nodes in the model graph and \( |N_p| \) denotes the number of nodes in the pattern graph.

Ullmann's algorithm [164] (based on a matrix representation of graphs and a backtracking pro-
cedure) is probably the most widely used generic algorithm for the subgraph isomorphism problem.
However, such general algorithms (like Ullmann's) are typically too general to be used in graph trans-
formation tools, i.e., it is hard to exploit the domain specific restrictions during the pattern matching
process.

In case of graph transformation systems, the pattern graph is constituted from the LHS of a rule
(and possibly from the negative condition graphs), while the model graph is the graph representation
of our system model. Typically, – even in complex model transformations – a pattern graph is relatively
small (with less than 20 nodes) while the model graph can be relatively large (with ten to hundred
thousands of nodes). Moreover, the graph representation of real systems are typically sparse, i.e., the number of edges are linear in the number of nodes.

Since, typically, we have an a priori fixed set of graph transformation rules, the complexity of a graph transformation step induced by this set of rules becomes polynomial (as \( |N_p| \) is now a constant).

**Modifications**

In contrast to graph pattern matching, the modifications prescribed by a graph transformation rule can be performed efficiently, in linear time provided that a matching is already constructed successfully.

- For *deletion*, we only have to select those nodes and edges in the rule that are part of the LHS but they do not belong to the RHS, and based on the matching, we easily select the corresponding model elements that should be removed.
- For *addition*, we simply create fresh nodes and edges and connect them according to the RHS of the rule.

As a consequence, the critical part of any implementation of graph transformation systems is an efficient graph pattern matching phase.

### 5.1.3 Graph pattern matching strategies

For space considerations, below we only provided a brief conceptual overview on different graph pattern matching strategies used in existing graph transformation systems. A detailed overview of these algorithms can be found in [158].

These algorithms can be grouped into two main categories (called as *strategies* in the sequel): (i) graph pattern matching based on constraint satisfaction, and (ii) algorithms based on local searches with heuristics.

**Algorithms based on the Constraint Satisfaction Problem (CSP)**

The constraint satisfaction problem (CSP) [100] is to find possible values for instantiating a group of variables while some conditions (denoted as constraints) are fulfilled.

**Definition 5.1.** A *constraint satisfaction problem* (CSP) is defined by an ordered set of variables \( X = (x_1, x_2, \ldots, x_n) \), a finite domain \( D_i \) of values for each variable \( x_i \), and a set of \( C \) constraints. A constraint \( R_{j_1, \ldots, j_r} \) is a subset of \( D_{j_1} \times \cdots \times D_{j_r} \) and restricts the values of the variables \( x_{j_1}, \ldots, x_{j_r} \) to be in that subdomain.

The assignment of values to variables is said to be *complete*, if all variables in \( X \) are included. An assignment is *consistent*, if it satisfies all the constraints. Note that an incomplete assignment may be consistent, and vice versa, an inconsistent assignment might be complete, thus completeness and consistency are independent.

The solution for a CSP is to find one or all the assignments that are complete and consistent. Instead of systematically generating all possible value combinations for the variables and testing them for constraint satisfaction, CSP algorithms gradually refine the domains of variables until they contain a single value, which is a consistent solution for the CSP problem. When a domain is reduced by eliminating a value or instantiating a variable (in a certain order) consistency checks are performed repeatedly. If this process (called *constraint propagation*) detects that the reduced domain becomes inconsistent, backtracking (or backjumping) is performed, and we start to investigate a new value of the domain. Since each value eliminated from a domain exponentially reduces the search space, constraint propagation combined with backtracking can be very efficient in practice.
5 Automated Model and Program Generation for Transformations

When interpreting graph pattern matching as a CSP problem, (i) the nodes of the pattern graph are the variables, (ii) the nodes of the model graph constitute the domains of these variables, and (iii) the edges of both graphs are the constraints of the CSP problem (provided that edges are relations on nodes as in our case). Additional constraints prescribe that the model graph should be type consistent with the pattern graph.

The pattern matching of the AGG [104] graph transformation tool is primarily based on CSP algorithms.

Local searches with heuristics

A different strategy based on local searches with heuristics [202] is used in the PROGRES [151] and, especially, in the FUJABA [114] systems. Instead of implementing a complex graph pattern matching engine which is parameterized with the pattern graph and the model graph (interpreted approach), a pattern dependent search plan is generated (compiled approach), which initiates local searches on the model graph.

In PROGRES, first an action graph is generated which consists of the pattern graph (i.e., the LHS of the rule) and so-called action nodes and edges. Action nodes of two major classes are created.

- **Test nodes** are used for verifying if some condition, which is necessary for a match is satisfied.
- **Get nodes** retrieve one or more elements (nodes, edges, sequence of edges) from the model graph which might match the corresponding left side element.

These action nodes are linked to LHS nodes by $m$ (match) or $r$ (require) edges. Those LHS nodes which are linked to an action node by an $m$ edge are matched after a successful execution of that action provided that all LHS nodes marked with $r$ edges leading from the current action node have already been matched beforehand.

Note that more than one action node can match some elements. This redundancy allows to select an optimal matching plan from a set of search plans of different costs.

After this, a sequence of action nodes will be called a valid search plan if all the actions only require elements that are matched by previous actions in the sequence. A plan is complete, if all LHS elements are matched by one of the actions involved. It is proved in [201] that valid search plans can always be found if there is a matching pattern. Therefore, the key of efficient pattern matching is to find the optimal search plan for a given action graph. Since this is still computationally hard, in [201] the authors proposed several heuristics to efficiently solve the problem for many practical cases.

For each kind of action an estimated cost is calculated. This cost usually smaller for test actions than for get actions, since it is evidently easier to test the existence of some nodes and edges (e.g., by using a hashtable) than to query the graph database (even if it is indexed). In this way, a new decision (to match an element in the model graph to an element in the pattern graph) is postponed until all checks are performed. A simple greedy algorithm then creates the optimal search plan according to these costs.

Based upon similar principles, the FUJABA system [114] compiles an object-oriented version of simplified search plans where the matching pattern is searched locally by navigating through links between objects.

Meanwhile, from a strict theoretical point of view, global CSP strategies can be more efficient than local searches, practical experiences in the previous graph transformation systems demonstrated that pattern matching by navigation is also very efficient in most practical cases, moreover it fits better to the object-oriented nature of UML.

5.1.4 Automated code generation approaches

Code generation has recently become a hot topic in an industrial asset. Several vendors are developing code generators that yield a part of the source code as implementation for the target application.
The majority of such code generators only consider to generate a skeleton of the static structure of the system (i.e., class and package declarations). Since the model transformation process is dynamic, these code generators are insufficient for our purpose. On the other hand, most sophisticated code generators also support the generation of dynamic code regions (i.e., implementation for statecharts, sequence diagrams as in case of UML). Dynamic code generators will be denoted as program generators in the sequel to distinguish them from traditional static code generators.

Another problem with typical code generators is that the code generation mechanism is hard-wired in the tool, i.e., new code generators can only be developed by the vendors by modifying the code generator program. This has a certain drawback from a system architect’s point of view, namely, he is not allowed to perform even minor modifications to tailor the code generator to his or her specific problem.

Model transformations necessitate the customizability of the code generation process. But this customizability cannot be achieved without allowing the system architects to control the program generation process, which can become very error prone without a well-founded program generation methodology.

**Program generators in an MDA environment**

Customizability of code generators has also become extremely important in MDA. According to MDA, platform specific models (PSMs) are captured on the UML model level, and all of such PSMs are derived from the platform independent model (PIM) by appropriate model transformations. Afterwards, the code generators are developed for the separate platforms, and they take the actual system model as parameters.

Unfortunately, MDA (as it is currently) does not specify how these code generators themselves can be designed and implemented possibly in a standard and mathematically well-founded way. Therefore existing customizable code generators (like the iUML [95] or StP [9] tools, or solutions based on the already discussed XSLT methodology) have necessarily chosen an own solution for describing the code generation process. In iUML, the code generation process is also considered to be a special dynamic system, which is thus captured by Action Semantics expressions [123]. Meanwhile, StP offers a new language that allows a pattern based code generation process, but with a very severe restriction, namely, that the UML model itself cannot be modified by the code generator. This introduces the same kind of problems already discussed in XSLT transformations. As a conclusion, both approaches are too low-level as a specification method of code generators, moreover, their theoretical well-foundedness is also problematic.

While the code generation problem is very relevant in an MDA environment, we restrict our investigations to automated program generators developed for model transformations (which is still a very general subproblem). We believe that our proposal would be useful in the general code generation process as well.

**Problem statement**

Although many existing model transformation approaches tackle the automated generation of transformation programs and provide a precise mathematical and conceptual background for them, all these solutions raise two major problems.

1. Only research prototype tools exist for supporting such transformations, which might not scale up in industrial applications.
2. The automated program generation method of such tools is typically low-level, hard-wired, and error prone.
5.2 Automated Program Generation for and by Model Transformations

We propose a reflective method for the automatic generation of the implementation for transformation systems specified by a set of graph transformation rules and a control flow graph. The program generator takes a UML profile tailored to model transformation systems as the input, and produces the output Prolog program by successive model transformation steps. In this respect, only the core of the program generator is implemented by hand, and afterwards, this core provides automation for additional features of the VIATRA model transformation system.

The automatic program generation approach consists of two major parts.
5.2 Automated Program Generation for and by Model Transformations

- As the specification of model transformation is expressed at a very high level, **transformation specific parts** of the automatically generated program must bridge a huge abstraction gap. Bridging such a gap is always at a high risk for generating erroneous code. Moreover, the generated code might implement similar structures more than once thus introducing undesired redundancy to the system.

- To increase the abstraction of programs (and decrease in turn their redundancy), the basic **control structures** (rule application modes) are required to be identified and collected into a separate module. In this respect, the automatically generated program is just a skeleton, which calls these common routines by proper parameters. Thus, only a rather high-level code is needed to be generated, while the single instructions are performed by the common module. These auxiliary predicates will also be denoted as **common code predicates** in the sequel.

Let us examine the properties of transformation rules and control structures from such a point of view whether transformation specific or common structures should be dominating in an automated implementation.

- For generating the Prolog code for a model transformation rule, the transformation specific parts will be dominating, as the sequence in which the objects of the LHS have to be matched (which is the most crucial step regarding efficiency) is highly dependent on the rule itself. Moreover, one cannot tell in advance what modifications are prescribed by the rule.

- In contrast to rules, major control flow structures are included in the common module, as the underlying algorithmic skeletons are similar. In this respect, common Prolog predicates will be equipped with rules and transformation units as parameters.

The following sections are thus concerned with (i) an overview of the Prolog data structures used in transformations; (ii) the automatic program generation for rules using the previous structures; (iii) the module for implementing the virtual machine that executes a macro step in the control flow graph built upon algorithmic skeletons of rule application modes.

### 5.2.1 Prolog representation of model transformation rules

#### The graph model

Model transformations manipulate on reference models, which are graphs constructed in correspondence with their metamodels. Now, these models are transformed into a Prolog term representation and stored as dynamic predicates in an internal fact database, which can be modified arbitrarily at run-time.

**Proposal 5.2 (Representing models in Prolog).** The correspondence between objects (graph nodes) and Prolog terms are characterized by the following rules (derived directly from the model level ASM representation of VPM models, cf. Sec. 3.5).

- From a model object of type **type** with an identifier **id**, the predicate **type(id)** is generated (**type** and **id** are considered to be Prolog atoms)
- From a model link of type **type** with its own **id**, source **src** and target **trg** identifiers, the predicate **type(id, src, trg)** is generated.
- From a model graph attribute (attached to the node identified by **nid**) with a name **name**, and having value **value**, the predicate **name(ownid, nid, value)** is generated, where **ownid** is a unique identifier referring to the attribute (generated automatically).

Each model (either source, target or reference) is stored in a distinct Prolog module. In this way, they can easily be accessed, yet, they are kept separated from each other. When using Prolog modules, a term of a model can be accessed by prefixing the term with the identifier of the model.
Example 5.3. For instance, for UML statecharts (see the metamodel in Fig. 8.7) we are able to access simpleState(s1) of model sc (stored in a module with identical name) by sc:simpleState(s1).

Matching graph patterns

Graph pattern matching is implemented by using the powerful unification mechanism of Prolog. In rule graphs, the identifiers of nodes and edges are normally variables, which variables get instantiated during the pattern matching process. If a variable is instantiated, it serves as a precondition for all the terms accessed later, which contain this specific variable.

Example 5.4. Let us now consider the LHS of rule \textit{valid2plaeR} (see Fig. 8.16) taken from the EHA2PN transformation in Section 8.3.

It prescribes the presence of a \texttt{hEvent} \texttt{E} accepted as a valid event by a \texttt{hQueue} \texttt{Q}. In the Prolog representation, such a query would look like the following (all the identifiers with capital initials are variables, thus instantiated when applying the rule).

\begin{verbatim}
  eha:hQueue(Q), % selecting a hQueue Q
  eha:validEv(E1,Q,E), % selecting a related hEvent E
  eha:hEvent(E), % testing the type conformance of E
\end{verbatim}

All the terms are matching a corresponding class or association (node or edge type), while the Prolog variables and graph object identifiers are similarly denoted. Although, such a representation of LHS queries might be sufficient at first sight, unfortunately, they raise several problems concerning efficiency and the handling of abstract classes.

- When the unification of a Prolog term \textit{may} succeed multiple times, a choice point is generated, and all the matching terms can be enumerated by backtracking. However, when the execution of the previous program skeleton arrives at to unify \texttt{hEvent}, its variable \texttt{E} is already instantiated, thus (as graph object identifiers are considered to be unique) this call cannot succeed more than once. As a result, an unnecessary choice point is generated, which decreases the efficiency of the program when backtracking is required.
- Similarly, the match of attribute terms can never succeed more than once.
- Graph patterns in the LHS of a rule may contain abstract nodes, i.e. nodes with a corresponding abstract metamodel class. A pure syntactical transformation from LHS patterns to Prolog terms would be unable to handle such matches.

To avoid the previous problems, all the terms in Prolog programs (just in programs not in the model modules) are embedded as a parameter into another predicate, which is responsible for properly handling the matching of nodes and edges.

A simplified implementation of the \texttt{node} and \texttt{edge} clauses (disregarding from type checking) is as follows.

\begin{verbatim}
node(Term):- call(Term).
node1(Term):- call(Term), !.
edge(Term):- call(Term).
attr(Term):- call(Term), !.
\end{verbatim}

Example 5.5. The needed extensions for the \textit{valid2plaeR} example (see Fig. 8.16 in the EHA2PN transformation in Section 8.3) are the following:

\begin{verbatim}
node(eha:hQueue(Q)), % choice point generated
edge(eha:validEv(E1,Q,E)), % choice point generated
node1(eha:hEvent(E)), % no choice point generated
\end{verbatim}
5.2 Automated Program Generation for and by Model Transformations  113

Please note the different predicates (\texttt{node} and \texttt{node1}) for accessing nodes, from which \texttt{node1} allows at most one successful match by applying the cut symbol after successfully calling for the \texttt{hEvent} state \texttt{E}. Naturally, a similar distinction can be used for edges but it is unnecessary for attributes (attributes are always cut after being matched). The \texttt{node} (and \texttt{node1}) predicate is also responsible for accessing the indirect "instances" of abstract classes.

**Modifying the models**

In order to allow the consistent modification of the Prolog representation of models, we basically have to encode the VPM operations introduced in Sec. 3.4 in Prolog, which is rather straightforward. Since Prolog has meta-programming facilities, a single predicate \texttt{add/1 (del/1)} is sufficient for addition (deletion) where the clause to be added (removed) is passed as an argument.

*Example 5.6.* Continuing our \texttt{valid2placeR} example, the additions prescribed by the RHS of the rule take the following form.

\[
\begin{align*}
\text{add(node}(\text{pn:place}(P))) & , & \% \text{ adding a Place node to the PN model} \\
\text{add(node}(\text{ref:refQEvent}(R))) & , & \% \text{ adding a RefQEvent node to the Ref model} \\
\text{add(edge}(\text{ref:src}(C1,R,Q))) & , & \% \text{ adding an edge to the ref model} \\
\text{add(edge}(\text{ref:src}(C2,R,E))) & , & \% \text{ adding an edge to the ref model} \\
\text{add(edge}(\text{ref:trg}(C3,R,P))) & , & \% \text{ adding an edge to the ref model}
\end{align*}
\]

The first a single node clause (\texttt{pn:place(H)}) is added to the \texttt{eha} model. Then we add a node to the reference metamodel (\texttt{ref:refQEvent(R)}), and connect it to the nodes of the source and target model by corresponding edges (\texttt{ref:src}(C1,R,Q), \texttt{ref:src}(C2,R,E) and \texttt{ref:trg}(C3,R,P)). The predicate \texttt{add} (an auxiliary predicate) is responsible for generating a unique identifier for objects before adding them to the database.

**Negative application condition**

Negative application conditions prohibit the presence of a matching pattern, thus all the possible extensions of the LHS need to be investigated. If the negative pattern is found then the application of the rule should fail, otherwise, if there is no occurrence of the negative pattern, the application of the rule should continue with additions and deletions.

*Example 5.7.* Let us consider now \texttt{compStateR} rule (Fig. 5.2) as an example for negative conditions (taken from the SC2EHA transformation of [178]).

\[
\begin{align*}
\text{LHS} & \quad \text{RHS} \\
\text{S: CompState} & \quad \text{S: CompState} \\
\text{substates} & \quad \text{src} \\
\text{PAR: CompState} & \quad \text{RefState} \\
\text{isConc=T} & \quad \text{trg} \\
\end{align*}
\]

\[
\begin{align*}
\text{compStateR}
\end{align*}
\]

*Fig. 5.2. Rule compStateR from [178]*

The corresponding Prolog code should resemble to the following piece of code (note that \texttt{\rightarrow} is a syntactic sugar for the cut operator used basically in if-then-else structures of Prolog). The entire clause fails if and only if all the four calls succeed.
node1(sc:compositeState(S)), % LHS pattern
( edge(sc:subvertex(E1,PAR,S)),
  node1(sc:compositeState(PAR)),
  attr(sc:isConcurrent(A2,PAR,'true')) ->
  fail % fail if succeeded
  ;
  true % continue otherwise
), ...;

Note that objects appearing both in LHS and the negative parts are only included in the former to obtain a more efficient implementation.

The example clearly demonstrates that the generated program partially contains transformation dependent (translated) Prolog code (i.e., the sequence of terms representing queries on the model graph) and (interpreted) calls to built-in auxiliary routines from a VIATRA library (like node(), edge()), and routines implementing different modes of rule application discussed in the upcoming section.

5.2.2 Implementing the control structures in Prolog

The implementation of the virtual machine that executes the control flow graph (CFG) of a model transition system builds upon the reflective property of program generation in VIATRA as the operational semantics of this virtual machine is also defined by model transition systems. In this respect, the program corresponding to a rule is generated automatically, and the implementation of the “meta” CFG (i.e., the CFG that defines the behavior of the virtual machine) is very simple. The entire semantics of the virtual machine consists of 11 rules, from which the handling of forall nodes is depicted in Fig. 5.3.

![Figure 5.3](image_url)

**Example 5.8.** The semantics of rule forallNodeR is as follows. When the virtual machine is to execute a step (pc = step), then the active CFG node in the active graph should be found and the selectR attribute of the associated graph transformation rule is set to true in order to select the rule for execution. Moreover, the program counter pc is set to exec indicating that now the virtual machine should execute the rule before making the next step and attribute rm (which represents the execution mode of a rule) is set to forall.

In the rule execution phase, the piece of code in the right region of Fig. 5.3 is called, which is responsible for executing the rule for all possible matches in the current state. This is obtained by causing an artificial backtracking whenever the rule application is succeeded. Finally, if all possible
matches are processed, the application of a rule in \textit{forall} mode is also successfully completed (including the case when there are no possible matches of the LHS).

Finally, the current node of the control flow graph is updated according to the success of rule application, and we proceed with the new current node.

Table \ref{tab:control_conditions} provides a brief overview on the implementation skeletons of the other control nodes in the CFG.

<table>
<thead>
<tr>
<th>Control condition</th>
<th>Prolog code</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule1, rule2</td>
<td>rule1, rule2</td>
</tr>
<tr>
<td>try(Rule)</td>
<td>try(Rule):-</td>
</tr>
<tr>
<td></td>
<td>call(Rule), !.</td>
</tr>
<tr>
<td>nondet(rule1; rule2)</td>
<td>( rule1</td>
</tr>
<tr>
<td></td>
<td>; rule2</td>
</tr>
<tr>
<td></td>
<td>), !.</td>
</tr>
<tr>
<td>loop(rule1)</td>
<td>loop(Rule):-</td>
</tr>
<tr>
<td></td>
<td>try(Rule),</td>
</tr>
<tr>
<td></td>
<td>loop(Rule).</td>
</tr>
<tr>
<td></td>
<td>loop(Rule).</td>
</tr>
<tr>
<td>forall(Rule)</td>
<td>forall(Rule):-</td>
</tr>
<tr>
<td></td>
<td>call(Rule), fail.</td>
</tr>
<tr>
<td></td>
<td>forall(Rule).</td>
</tr>
</tbody>
</table>

\textbf{Table \ref{tab:control_conditions}.} Implementing control conditions

- The \textbf{sequence} of two (or more) rules is implemented by the Prolog AND operator (comma). No additional code is required for the common code module. Sequence operation is used for selecting the next nodes defined by \textit{succeed} and \textit{fail} edges in the CFG.
- The \textbf{try (at most once)} semantics necessitates choice points generated by the successful application of the rule to be cut (the \textit{call} predicate of Prolog is a so-called meta-predicate for being able to call Prolog clauses passed as attributes).
- For the \textbf{non-deterministic choice} of two (or more) rules, the Prolog OR operator (semi-colon) is used without calls to additional auxiliary routines.
- The clause for the \textbf{loop (as long as possible)} semantics of a rule tries to apply the rule first, and calls itself recursively, if the rule were able to be applied. The termination of this control condition cannot always be guaranteed, as it depends on the successful application of the rule.
- The \textbf{forall} semantics enumerates all the possible matches by causing artificial backtracking (using the always unsuccessful \textit{fail} clause). As the \textit{forall} application of a rule is successful even if the rule cannot be applied at all, a second \textit{forall(Rule)} clause is needed to guarantee that property. This control condition always terminates.

\subsection{Automated program generation by model transformation}

The automated program generation of model transformation scripts allows the transformation designers to focus on the \textit{design} of a model transformation rather than the \textit{implementation}. Previous experiments (in project HIDE \cite{[24]}) demonstrated that the quality of an automatically generated executable transformation program is much higher than a manually written target program. Moreover, once the automated program generator is completed, the time and workload related to the design of a single transformation is drastically decreased.

Such a program generator receives a high-level description of model transformation rules as the input, and it should generate the corresponding Prolog program from this specification that implements
the transformation. Our proposal is to regard the problem as a model transformation task having the description of graph transformation rules as the source model and the Prolog code as the target model.

Therefore, the program generation process of model transformation rules should be divided into several intermediate steps (as depicted in Fig. 5.4).

1. Model transition systems are specified in a UML CASE tool (Rational Rose was used for our experiments), and exported in the standard XMI format. At this stage, rules are graphically represented by a special structure of packages, classes and associations with stereotypes.

2. This UML model is transformed into a GraTra model conforming to a metamodel of graph transformation systems. This GraTra description (with nodes, edges, attributes etc.) is identical with the transformation rules used previously.

3. In VIATRA, model graphs are represented as predicates in a fact database. For this reason, the previous GraTra model is projected into a Logic model containing a sequence of predicates for each rule. This phase also contains an optimization process concerning the ordering of LHS query predicates for improving the efficiency of transformations.

4. The bridge between visual (graph based) and the textual language of Prolog is provided by the parse tree of the Prolog code. In this sense, the Logic model is transformed into a corresponding Prolog parse tree, and the target Prolog code is printed out by traversing this tree in an in-order way.

These intermediate steps provide several advantages compared with a direct transformation approach.

- There is a huge abstraction gap between a visual UML-based specification of the transformation and even such a high-level programming language as Prolog. Thus splitting the transformation into several subtransformations decreases the complexity of the individual steps, which eases not only the implementation but also the verification of the automated program generation.

- The use of intermediate models increases reusability. For instance, when generating the input language of a model checker for the verification of a model transformation, only the final step needs to be altered.

- The intermediate GraTra model provides the right basis for generating the upcoming standard XML description of graph transformation systems [157] by a simple transformation from the GraTra XMI format to GXL/GTXL.

- The final metamodel of the transformation is a simple tree structure based on the BNF representation of the Prolog code (or more generally speaking, by some Chomsky grammars), containing only the notions of terminals and non-terminals as classes. Terminal nodes are enriched with the attached text attributes used for storing the pieces of code. At the final stage, the code generation process simply traverses this code tree and prints the text values stored at each terminal node that is reached. This graph traversal algorithm is general in the sense that same algorithm can be used for different programming languages.
In fact, each intermediate transformation step is specified (and implemented) as model transformation, which means that the entire code generation process is captured by graph transformation. In this sense, the implementation of model transformation rules is specified by model transformation rules. This approach is similar to the bootstrapping process of compiler design, where, for instance, a C compiler is written in C and compiled by an existing C compiler, and recompiled by itself afterwards to provide a more efficient and reliable target code. In VIATRA, the current version of the program generator is implemented manually, while future versions (with additional features, and more efficient / reliable target code) are generated by using the existing version of the program generator.

In the following, the program generation process for model transformation rules will be demonstrated on the code generation for a small sample rule (simpleStateR of Fig. 5.5 taken from [178] and extended with copying the names of states).

UML representation of rules

Describing the UML representation of graph transformation rules in VIATRA is out of the scope of the current thesis. However, a brief overview is provided in Fig. 5.6.

Basically, a graph (LHS, RHS, or Neg) is represented by UML Packages, rule nodes are denoted by UML Classes, while UML Associations depict edges in rules. Therefore Fig. 5.6 is the VIATRA representation of simpleStateR of Fig. 5.5.

We must admit that the UML notation used in the FUJABA tool provides a closer correspondence with the UML philosophy as FUJABA rules are basically collaboration (object) diagrams [66]. Unfortunately, leading UML CASE tools (like Rational Rose) do not sufficiently support object diagrams. Therefore, in order to obtain an open architecture, which is as independent of particular CASE tools as possible, we decided on the use of UML extension mechanisms (stereotypes) to attain the same modeling effect.

The metamodel of graph transformation

Although, model transformation rules are based upon the paradigm of graph transformation, it can be regarded as an ordinary modeling language (like statecharts or automata), thus, its metamodel (Fig. 5.7) can be created by strictly following its definitions (Def. 4.1).
In fact, this metamodel also served as the basis of the Budapest proposal [189] submitted as a response to the standardization calls from the graph transformation community, which highly influenced the upcoming standard [157]).

![Diagram of the metamodel of graph transformation](image)

**Fig. 5.7.** The metamodel of graph transformation

- The metamodel expresses that a graph transformation rule is composed of a *lhs* and an *rhs* graph, and a mapping *lhs2rhs* between objects in the LHS and RHS. Negative application condition graphs can be handled similarly, however, they are not handled below for space considerations.
- A *Graph* is composed of abstract *GraphElements*, which are either instances of a *Node* or an *Edge*. They have several attributes, such as *varID* for storing the identifier, *type* for the type of the object and *model* for identifying the model the object belongs to.
- The instances of the class *Attribute* can be attached to *GraphElements*, which class in turn contains two MOF attributes *name* and *value* (note that the notion of attribute is overloaded).
- A *MapElem*, which is embedded into a *Mapping*, contains a reference (*mapsFrom*) to an LHS object, and a reference to an RHS object (*mapsTo*). An additional semantic constraint is needed at the current point to prescribe that a map element must not connect LHS nodes to RHS edges and vice versa.

**Example 5.9.** The graph model of the transformation rule *simpleStateR* is depicted in Fig. 5.8 in order to give an impression about the entire program generation process. Edges of type *contents* are printed with dashed lines to improve the clarity of the relatively complicated figure.

Note that this graph based representation does not explicitly contain any information on the optimal pattern matching of the LHS, thus such an ordering information needs to be added later during the forthcoming model transformation.

**The term representation of rules**

The second model transformation generates a term representation for graph nodes and edges. Moreover, a nearly optimal ordering of the query predicates of the LHS (nearly optimal from the point of view of graph pattern matching) is also provided.

- The metamodel of predicates (see Figure 5.9) is composed of *Clauses* on the top of the hierarchy.
- The clause is constituted from list of facts (*Facts*), one for specifying the predicates of the LHS query, one for the prescribed additions of predicates and one for deletions (in a more detailed metamodel, a fourth *Facts* would contain predicates for the negative condition).
- A FactLs object is built up from Facts (an abstract class), which is either a NodeFact, an EdgeFact or an AttrFact. The facts are information preserving in the sense that e.g. links between nodes end edges (see from and to relations in the GraTra metamodel) are now encoded into attributes fromID and toID.

- However, new connections have been introduced to represent the sequence in which these facts are to be generated in the code. The first and the successor predicates in the sequence are identified by first and next edges.

**Example 5.10.** The model graph of the predicate representation of our running example (the implementation of rule simpleStateR) is depicted in Figure 5.10.

The objects of the LHS (node n1 and attribute a1) are directly transformed into predicates, while a predicate representation contains only those parts of the RHS graph, which cannot be mapped to a LHS object (such nodes are n2, n3, e1, e2 in the example). In this way, the predicate representation is more compact when compared to the corresponding GraTra description.
The ordering of predicates is the most crucial part of the transformation with respect to run-time performance. Informally, graph structures that are visited and matched each nodes in the LHS (and NEG) parts have to be ordered into a sequence with optimal run-time.

**Proposal 5.11 (Optimizations in clause ordering).** The clause ordering algorithm implemented in VIATRA is as follows.

- The sequence of query predicates *commences with* the check of objects obtained as *rule parameters* since the images of such nodes and edges are fixed in the host graph.
- If a rule (such as `simpleStateR`) contains no parameters, *an arbitrary node can be matched first*. However, further optimization is possible here to select first that class of nodes which has the least number of instances in order to obtain a search tree having the least number of branches from the root.
- After node has been matched, all of its *attributes are collected and checked*.
- Afterwards, the LHS graph is traversed by *matching an edge leading from (or to) a previously matched node* (checking in turn its type), which step selects the target (source) node as well. Then we merely check whether *the type of the currently identified target (source) node in the host graph corresponds to its LHS graph specification*. Please note that type checking against a metamodel is a less expensive operation than matching graph objects of the LHS.
- Finally, the LHS graph traversal algorithm continues with un-traversed edges. If the LHS graph is composed of more than one components then the algorithm is required to be applied for each component.

The RHS objects are ordered differently in order to maintain the invariant property that the application of a rule always result in a well-formed graph (without dangling edges).

- The deletion of edges should always proceed the removal of nodes. Whenever a node fact is required to be deleted, all the edges connected to that specific node and attributes attached have to be implicitly deleted at the same time. Thus, the remove operation is partially implemented by built-in VIATRA routines.
In contrast to deletions, the order of adding graph objects is just the opposite. A correct order should start with the addition of nodes followed by the construction of attributes, and finally, the creation of edges. Keeping the specific order ensures that the result is always a graph.

**Generating a parse tree**

From a predicate representation of graphs, the generation of Prolog facts does not require the bridging of a huge abstraction gap, especially in such a case, when the order of predicates has already been determined. Each syntactic element of Prolog is transformed into terminal graph nodes (controlled by the grammar of Prolog).

However, in order to keep our program generation approach language independent, not the final metamodel (in Figure 5.11) is the metamodel of Chomsky grammars, and not the metamodel of Prolog.

![Fig. 5.11. The metamodel of parse trees (grammars)](image)

The advantage of such a general solution originates in the fact that only a single graph traversal algorithm is required for the final code generation step for any programming language, while the well-formedness of the parse tree can be verified against traditional context-free grammars. Thus, for this final step, the grammar of the programming language is also needed. A highly simplified grammar of Prolog is printed below.

```
Program ::= Clause Program | Clause
Clause ::= Pred ',' Terms .'.' | Pred.
Pred ::= atom '(' Arg ')' | atom
Arg ::= Pred | var | atom
Terms ::= Pred ',' Terms | Pred
```

**Example 5.12.** The predicate representation of our rule `simpleStateR` is transformed into a parse tree. Figure 5.12 shows a meaningful part of this tree containing subgraph for the code being generated for the addition of the `hState T1`. (This time the values of `text` attribute of terminal nodes and the names of nonterminals are printed inside the graph node instead of node identifiers.)

The top (depicted) level of the parse tree consists of the `Pred` nonterminals separated by commas as terminals. One level below, the addition is specified by the predicate `add`. This predicate contains another predicate as attribute, which describes the fact `(eha:hState(T1))` to be added to the database.

The interested reader may check that this tree is a well-formed (part of a) Prolog program by parsing this tree against the grammar and the metamodel.

**The generated Prolog code**

The parse tree is traversed by the following simple algorithm (which visits the tree in a top-down, left-to-right order) in order to generate the final textual representation of rules (i.e. the Prolog code).

1. Start from the root non-terminal of the tree.
2. If a terminal node is reached then print the value of its `next` attribute.
3. If a non-terminal node is reached then
   a) visit the node identified by the `first` edge (i.e. apply the algorithm recursively from (2))
   b) while a `next` edge leads from the current node apply the algorithm recursively from (2) to the next node

**Example 5.13.** The algorithm yields the following code when applied to the complete parse tree generated by the previous model transformation step.

```
simpleStateR :-
  % LHS
  node(sc:simpleState(S1)),
  attr(sc:name(A1,S1,N)),
  % RHS
  add(node(ref:refState(R1))),
  add(node(eha:hState(T1))),
  add(attr(eha:name(A2,T1,N))),
  add(edge(ref:src(E1,R1,S1))),
  add(edge(ref:trg(E2,R1,T1))).
```

In this previous case, the final step of code generation was driven by a traditional algorithm. Alternatively, we may extend graph transformation rules with a code region which should be written as textual output when the rule is applied. Code regions would ideally offer a combination of graph transformation techniques with modern pattern-based code generators like StP [9] or XSLT [159], which provides a better fit to a model-driven development and code generation process. Naturally, further research is required in this area, although initial experiments with graph transformation rules enriched with code regions have been carried out in [158].

As a summary, the program generation process of model transformation rules was also designed by model transformations receiving a description of graph transformations as input and yielding the corresponding Prolog code as output.
5.3 Correctness of the Prolog Encoding

In the current section, we state that our Prolog encoding of model transformation rules is correct with respect to the ASM semantics of graph transformation. Since Prolog is a deterministic language while graph transformation is non-deterministic in general, this time the completeness of our approach cannot be established. For each Prolog computation generated from a model transformation system (MTS), we are able to show an equivalent run of the ASM of the original MTS, but we might have runs of the original MTS which cannot (always) be simulated in the Prolog encoding (i.e., whether it can be simulated or not in Prolog depends on the ordering of clauses).

Since B"{o}rger et. al. elaborated an ASM semantics of the full Prolog language in [31], a formal proof of correctness would also be possible. However, such a formal proof requires quite lengthy theoretical preparations, which is not directly related to the current thesis. Therefore, we decided to skip this formal proof for space considerations.

5.3.1 Correctness of the Prolog representation of graph transformation rules

First, we state that if the Prolog representation of a graph transformation rule (generated according to Sec. 5.2) is submitted as a query to the Prolog engine, the effects can be simulated by the original GT rule.

**Proposition 5.14 (Correctness of Prolog rules).** Let $\mathcal{B}^{pr}$ be the state of the Prolog database, and $\mathcal{F}(\mathcal{B}^{pr}) = \mathcal{B}^{ct}$ be its equivalent in the model level ASM representation of VPM models. Moreover, let $r^{pr}$ be the Prolog representation of a graph transformation rule $r^{ct}$ submitted as a query to the Prolog engine.

1. If the Prolog execution succeeds with a result state $\mathcal{B}^{pr}$ (denoted as $\mathcal{B}^{pr} \rightarrow \mathcal{B}^{pr'}$) then there exists a state $\mathcal{B}^{ct}$ such that $\mathcal{B}^{pr} = \mathcal{F}(\mathcal{B}^{ct})$ and $\mathcal{B}^{ct} = \text{next}_{r^{ct}}(\mathcal{B}^{ct})$.

2. If the Prolog execution fails, then there are no states $\mathcal{B}^{ct} \neq \mathcal{B}^{pr}$ such that $\mathcal{B}^{pr} = \text{next}_{r^{ct}}(\mathcal{B}^{ct})$.

Note that completeness might not be guaranteed in a case when there are several potential successful matchings of the LHS of a rule. Since Prolog is a deterministic language, the Prolog rule $r^{ct}$ can be applied on the same matching each time (depending on the dynamic database). However, if there are no potential applications of a Prolog rule, then we can conclude that the graph transformation rule $r^{ct}$ itself is not applicable.

5.3.2 Correctness of the Prolog encoding of control structures

For proving the correctness of the Prolog encoding of control structures, we have to investigate the resolution and backtracking mechanism of Prolog following [31].

A Prolog computation can be seen as systematic search of a space of possible solutions to an initially given query. The set of computation states is often viewed as carrying a tree structure, with the initial state at the root, and son relation representing alternative (single) resolution steps. Each Prolog computation state has to carry all information relevant for the computation state it represents. This information consists of the sequence of goals still to be executed, the substitution computed so far, and possibly the sequence of alternative states still to be tried.

When at a given computation node $n$ the selected literal (activator) $act$ is called for execution, for each possible immediate resolvent state a son of $n$ will be created, to control the alternative computation thread. Each son is determined by a corresponding candidate clause of the program, i.e. one of those clauses whose head might unify with $act$. All such candidate sons are attached to $n$ as a list $\text{cands}(n)$, in the order reflecting the ordering of corresponding candidate clauses in the program.

This action of augmenting the tree with $\text{cands}(n)$ takes place at most once, when $n$ gets first visited (in Call mode). The mode then turns to Select, and a resolution step is attempted, i.e., the
first unifying son from \textit{cands}(n) gets visited, again in \textit{Call} mode. The selected son is simultaneously
deleted from the \textit{cands}(n) list. If control ever returns to \textit{n}, (by \textit{backtracking}), it will be in \textit{Select} mode,
and the next candidate son will be selected, if any.

If none, that is if in \textit{Select} mode \textit{cands (n)} = [], all attempts at resolution from the state represented
by \textit{n} will have failed, and \textit{n} will be \textit{abandoned} by returning control to its father. This action is usually
called \textit{backtracking}. In case of the \textit{cut} operator (!), we can override the father of a computation node
by selecting a previous node in the computation sequence.

Now, we formalized the correctness of the Prolog implementation of the control structures. Note
that since the Prolog representation of a graph transformation rule (unlike the Prolog representation of
control structures) generates choicepoints for all querying \textit{node1/1} and \textit{edge3/3} clauses (while cutpoints
are generated for \textit{node1/1} and \textit{attr/3}), the main task of the proofs is to take care of such choicepoints
(and cutpoints).

\textbf{Proposition 5.15 (Correctness of \textit{try}).} If \textit{r} is a correct implementation of \textit{r} (see Prop. 5.14
above), then \textit{try}(\textit{r}) (see Table 5.1) is a correct implementation of \textit{try}(\textit{r}) (see Def. 4.4.2).

Formally, let \textit{A} be the state of the Prolog database, and \textit{F}(\textit{A}) = \textit{A} be its equivalent in the
model level ASM representation of VPM models, and let \textit{try}(\textit{r}) be the query submitted to the Prolog
engine. If the Prolog execution succeeds with a result state \textit{B} (denoted as \textit{A} = \textit{B}) then there
exists a state \textit{B} such that \textit{B} = \textit{F}(\textit{B}) and \textit{B} = \textit{next}_{\textit{try}(\textit{r})}(\textit{A}).

\textbf{Proposition 5.16 (Correctness of \textit{nondet}).} If \textit{r} is a correct implementation of \textit{r}, and \textit{r} is a correct implementation of \textit{r}, then \textit{nondet}(\textit{r}, \textit{r}) (i.e., \textit{r}; \textit{r}) is a correct implementation of \textit{nondet}(\textit{r}, \textit{r}).

Formally, let \textit{A} be the state of the Prolog database, and \textit{F}(\textit{A}) = \textit{A} be its equivalent in the
model level ASM representation of VPM models, and let \textit{nondet}(\textit{r}, \textit{r}) be the query submitted to the Prolog
engine. If the Prolog execution succeeds with a result state \textit{B} (denoted as \textit{A} = \textit{B}) then there
exists a state \textit{B} such that \textit{B} = \textit{F}(\textit{B}) and \textit{B} = \textit{next}_{\textit{nondet}(\textit{r}, \textit{r})}(\textit{A}).

\textbf{Proposition 5.17 (Correctness of \textit{sequential application}).} If \textit{r} is a correct implementation of \textit{r}, and \textit{r} is a correct implementation of \textit{r}, then \textit{seq}(\textit{r}, \textit{r}) (or, in Prolog syntax, \textit{r}, \textit{r}) is a correct implementation of \textit{seq}(\textit{r}, \textit{r}).

Formally, let \textit{A} be the state of the Prolog database, and \textit{F}(\textit{A}) = \textit{A} be its equivalent in the
model level ASM representation of VPM models, and let \textit{seq}(\textit{r}, \textit{r}) be the query submitted to the Prolog
engine. If the Prolog execution succeeds with a result state \textit{B} (denoted as \textit{A} = \textit{B}) then there
exists a state \textit{B} such that \textit{B} = \textit{F}(\textit{B}) and \textit{B} = \textit{next}_{\textit{seq}(\textit{r}, \textit{r})}(\textit{A}).

\textbf{Proposition 5.18 (Correctness of \textit{loop}).} If \textit{r} is a correct implementation of \textit{r}, then \textit{loop}(\textit{r}) is a correct implementation of \textit{loop}(\textit{r}) provided that \textit{loop}(\textit{r}) is terminating.

Formally, let \textit{A} be the state of the Prolog database, and \textit{F}(\textit{A}) = \textit{A} be its equivalent in the
model level ASM representation of VPM models, and let \textit{loop}(\textit{r}) be the query submitted to the Prolog
engine. If the Prolog execution succeeds with a result state \textit{B} (denoted as \textit{A} = \textit{B}) then there
exists a state \textit{B} such that \textit{B} = \textit{F}(\textit{B}) and \textit{B} = \textit{next}_{\textit{loop}(\textit{r})}(\textit{A}).

\textbf{Proposition 5.19 (Correctness of \textit{forall}).} If \textit{r} is a correct implementation of \textit{r}, then the rule \textit{forall}(\textit{r}) is a correct implementation of \textit{forall}(\textit{r}) provided that \textit{r} is independent of itself (i.e.,
the application of the rule will not disable potential matchings).

Formally, let \textit{A} be the state of the Prolog database, and \textit{F}(\textit{A}) = \textit{A} be its equivalent in the
model level ASM representation of VPM models, and let \textit{forall}(\textit{r}) be the query submitted to the Prolog
engine. If the Prolog execution succeeds with a result state \textit{B} (denoted as \textit{A} = \textit{B}) then there
exists a state \textit{B} such that \textit{B} = \textit{F}(\textit{B}) and \textit{B} = \textit{next}_{\textit{forall}(\textit{r})}(\textit{A}).
As a result, we showed that our Prolog encoding of model transformation systems is correct (with respect to the ASM semantics). In the upcoming section, we investigate how to provide more standard means to automatically implement model transformations.

5.4 UML Action Semantics for Model Transformation Systems

We describe a general encoding of model transformation systems as executable Action Semantics [109,123] expressions to provide a standard way for automatically generating the implementation of formal (and proven correct) transformations by off-the-shelf MDA tools. In addition, we point out a weakness in the Action Semantics standard that must be improved to achieve a stand-alone and functionally complete action specification language.

5.4.1 Action Semantics for UML: An overview

The **Action Semantics for UML (AS)** provides a standardized and platform (and implementation) independent way to specify the behavior of objects in a distributed environment. Basically, the user can describe the body of methods and executable actions in an abstract language prior to the implementation phase by constructing a dataflow like model.

**Action specification**

An action specification consists of the following elements (see the metamodel in Fig. 5.13).

- **Pins:** the input and output “ports” of an action having a specific type and multiplicity (a pin may hold a collection of values at a time if it is allowed by its multiplicity),
- **Variables:** an auxiliary store for results of intermediate computations,
- **Data flow:** connects the output pin of its source action to the input pin of the destination action thus providing an implicit ordering of action execution,
- **Control flow:** imposes an explicit ordering constraint for action pairs (predecessor–successor) having no connecting data flow,
- **Actions:** for object manipulation, memory operations, arithmetic, message passing, etc.,
- **Procedures** provide the packaging of actions with input and output pins, e.g., for method body. Procedures directly belong to objects.
Action execution

The execution of an action has the following stages in its life-cycle.

- **Waiting.** An action execution may be created at any time after the procedure execution for its containing procedure is created. On creation, an action execution has the status *waiting* and no pin values are available.

- **Ready.** An action execution with status *waiting* becomes *ready* on the completion of the execution of all prerequisite actions (that is all actions that are the sources of data flows or predecessors of control flows into the action becoming ready). The values of the input pins of the target action execution are determined by the values of the output pins from the prerequisite action executions via data flows.

- **Executing.** Once it is *ready*, an action execution eventually begins *executing* (the action semantics does not determine the specific time delay (if any) between becoming ready and actually executing).

- **Complete.** When it has finished execution, the action becomes *complete*. The action execution then has pin values for all output pins of the action computed according to specific semantics of different actions. After the output values from a completed execution have been copied, there is no longer any way for another execution to access the completed execution.

Actions have no default orderings (unlike sequential execution as in traditional programming languages); actions that are not implicitly ordered by data flow or explicitly ordered by control flow can be executed either parallelly or in an arbitrary order.

Types of actions

Specific semantics of different kind of actions can be grouped into the following main categories (only actions relevant for the encoding of Sec. 5.4.2 are enlisted).

- **Computation actions** (e.g. ApplyFunctionAction) are primitive actions for mathematical functions (not defined in the standard in details).

- **Composite actions** (see Fig. 5.14 for the metamodel) are recursive structures that permit complex actions to be composed of simpler ones providing means for basic control flow actions (e.g., LoopAction, ConditionalAction, GroupAction),

![Composite actions](image)

Fig. 5.14. Composite actions

- *GroupActions* simple collect subactions into a unit of higher abstraction level, and they may contain variables as well.
- **ConditionalActions** are composed of partially ordered Clauses, which consist of a (Boolean) test action and a body action. If the test action of a clause evaluates to true, we call it *enabled*. An enabled clause $c_1$ which precedes another enabled clause $c_2$ according to the partial order (imposed by the *predecessorClause* and *successorClause* relations) has a higher priority. The ConditionalAction thus finally executes the body action of an enabled clause with highest priority. If there are multiple clauses with highest priority, the clause to be executed is selected non-deterministically.

- **LoopActions** contain loop variables and a single clause. Loop variables are output pins, which may be (implicitly) connected to input pins of the clause. When executing a LoopAction, we execute the body action of the clause as long as the test action evaluates to true. After each micro step (execution of the body), the loop variables are refreshed.

- **Read and write actions** access, navigate, and modify model-level constructs (such as objects, links, attributes, slots, and variables). Since there is a large number of read and write actions, we only summarize the most important ones (without the metamodel).
  - *Accessing and manipulating variables*: ReadVariableAction accesses the value of a variable to store it on its output. AddVariableValueAction, on the contrary, saves the value of its input to a variable.
  - *Testing type conformance of objects*: ReadIsClassifiedObjectAction tests whether the object stored at its input pin conforms to a specific class, and retrieves a boolean value as result.
  - *Reading and writing objects*: ReadSelfAction retrieves its owner object to its output pin. CreateObjectAction generates a new objects of a specific type as result. DestroyObjectAction destroys the object stored at its input pin.
  - *Reading and writing links*: Given an object playing a certain role $r_1$ in a link, ReadLinkAction retrieves all the objects playing another role $r_2$ in the same link. CreateLinkAction creates a link, while DestroyLinkAction destroys the link of a certain type between two objects.
  - *Manipulating attributes*: Similar actions exists to access and manipulate attributes (omitted here for space considerations).

- **Collection actions** (such as FilterAction, MapAction, or IterateAction) apply a subaction to a collection of elements to avoid overspecification of control caused by explicit indexing and extracting of elements.
  - *FilterAction* keeps only those elements from the input collection in the result that satisfy a given condition.
  - *MapAction* executes the subaction in parallel for all elements in the input collection.
  - *IterateAction* executes a subaction to all input elements but it collects the intermediate results in a loop variable that might affect the further execution of the subaction.

**Syntax of actions**

The Action Semantics standard only defines a metamodel (and some well-formedness constraints) for the language without any restrictions on concrete syntax. In this respect, a well-formed action expression itself is a rather complex object diagram, which is easy to process for CASE tools but extremely hard to read and write system engineers. In fact, existing UML CASE tools with an integrated action specification language have their own textual notations for describing actions.

Therefore, the encoding of model transformation systems will be presented in the sequel on two levels: (i) first, in an own, self-explanatory pseudo action language to understand the overall idea of the encoding (instead of sticking to any specific existing dialects of AS tools), (ii) and then in a standardized way, by using object diagrams (to cope with AS technicalities).
5 Automated Model and Program Generation for Transformations

An ASM semantics for Action Semantics expressions

As Action Semantics is a semi-formal specification technique (where the semantics of the language is only informally defined), we have to define a precise formal semantics to the language in order to allow formal reasoning on the correctness of our encoding in Sec. 5.4.2–5.4.5. In Table 5.2, we briefly sketch a denotational (compiled) approach to such a formal semantics by deriving ASMs to an AS expression.

<table>
<thead>
<tr>
<th>AS</th>
<th>ASM</th>
<th>Informal explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure</td>
<td>μ(a) : ret</td>
<td>Rule</td>
</tr>
<tr>
<td>Action</td>
<td>μ(a) : ret</td>
<td>Rule</td>
</tr>
<tr>
<td>InputPin</td>
<td>r(a) : ret</td>
<td>Parameter of a rule</td>
</tr>
<tr>
<td>OutputPin</td>
<td>r(a) : ret</td>
<td>Return value of a rule</td>
</tr>
<tr>
<td>Variable</td>
<td>x</td>
<td>Nullary dynamic function</td>
</tr>
<tr>
<td>DataFlow</td>
<td>r_s(event)</td>
<td>Return value of the source rule is passed as parameter to the destination rule</td>
</tr>
<tr>
<td>ControlFlow</td>
<td>seq(r_pred, r succ)</td>
<td>The predecessor rule precedes (in a sequential execution) the successor rule</td>
</tr>
<tr>
<td>GroupAction</td>
<td>par r_i</td>
<td>parallel execution of subactions</td>
</tr>
<tr>
<td>ConditionalAction</td>
<td>if test(d_1) then body(d_1) else body(d_2)</td>
<td>if-then-else depending on the test result of the predecessor clause</td>
</tr>
<tr>
<td>ConditionalAction</td>
<td>if test(d_2) then body(d_2)</td>
<td>Independent if-then clauses</td>
</tr>
<tr>
<td>LoopAction</td>
<td>choose {R_i} with test(d_i) = true while test(d_i) do body(d_i)</td>
<td>execute the body of the clause while the test of the clause evaluates to true</td>
</tr>
<tr>
<td>ReadVariableAction(var)</td>
<td>var</td>
<td>access the variable</td>
</tr>
<tr>
<td>AddVariableValueAction(val, var)</td>
<td>var := val</td>
<td>update the value of variable var to val</td>
</tr>
<tr>
<td>ReadIsClassifiedObject(obj, cls)</td>
<td>instanceOf(obj, cls)</td>
<td>see Sec. 3.4.3 for definition</td>
</tr>
<tr>
<td>CreateObjectAction(obj)</td>
<td>obj := CreateObjectAction cls</td>
<td>create a new object obj from the reserve and then call the built-in VPM functions</td>
</tr>
<tr>
<td>DeleteObjectAction(obj)</td>
<td>deleteObj(obj)</td>
<td>delete object obj from the model using the built-in VPM function</td>
</tr>
<tr>
<td>Coll :=</td>
<td>Coll := {try</td>
<td>entity(src) \∈ entity(trg) &amp; \exists : (assoc(c, src, trg) &amp; assoc(c, trg, src))}</td>
</tr>
<tr>
<td>ReadLinkAction(src, assoc)</td>
<td>lk :=</td>
<td>create a new link identifier lk from the reserve and then call the built-in VPM functions</td>
</tr>
<tr>
<td>CreateLinkAction(obj_1, assoc, obj_2)</td>
<td>addConnection(linkFromAtoB(lk, obj_1, obj_2))</td>
<td>create a new connection between obj_1 and obj_2 from the model</td>
</tr>
<tr>
<td>DeleteLinkAction(obj_1, lk, obj_2)</td>
<td>delConnection(linkFromAtoB(lk, obj_1, obj_2))</td>
<td>delete link lk leading between obj_1 and obj_2 from the model</td>
</tr>
<tr>
<td>MapAction(Coll, subset)</td>
<td>forall c with c \∈ Coll do subset(c)</td>
<td>call the subaction subset in parallel for all elements of the input collection Coll</td>
</tr>
<tr>
<td>IterateAction(Coll, subset)</td>
<td>while Coll \≠ \emptyset do</td>
<td>call the subaction subset in one by one for all elements in Coll by accumulating the intermediate results</td>
</tr>
</tbody>
</table>

Table 5.2. ASM semantics of Action Semantics expressions

5.4.2 Action Semantics for model transformation systems: An overview

In the upcoming sections, we provide a general way to encode model transformation systems (MTS) into Action Semantics (AS) descriptions to provide a standard and platform independent way to implement transformations in the MDA environment.

Our generation technique takes the metamodel(s), and a model transformation system as the input, and generates a set of AS expressions as the output. The results of the transformations are obtained afterwards in the form of an object/collaboration diagram.
5.4 UML Action Semantics for Model Transformation Systems

A natural correctness criteria of the generation process would be that the execution of actions according to the Action Semantics standard should yield identical result with (at least one trace of) the formal generation process driven by the model transformation system itself starting from a given instance model. However, as the AS standard completely lacks any formal semantics, we have to define an ASM semantics for the AS expressions in order to formally reason about the correctness of our approach.

The overall idea basically follows the graph pattern matching techniques implemented in the PROGRES [151] and FUJABA [114] tools in procedural and object-oriented languages. The encoding consists of the following main steps (which steps will be introduced in details as “Proposals” later on in this section):

- **implementing graph pattern matching** by local searches in the user model based on collection actions and navigation capabilities of AS;
- **checking the non-existence** of certain objects and links prescribed by the negative conditions by a user defined function;
- **adding and deleting graph objects** by using actions for object and link manipulation;
- **implementing rule application modes** by various corresponding collection actions;
- **simulating the execution of the control flow graph** by conditional actions and explicit control flow restrictions.

The encoding will be introduced on our running example of the reachability problem of finite automata (of Fig. 4.10), which includes the handling of all these major problems.

### 5.4.3 Encoding the control flow graph

The handling of the control flow graph consists of modeling rule applications\(^1\) in a certain mode and defining the sequence of consecutive transformation steps.

**Proposal 5.20.** For each MT rule applied in loop or forall mode, a GroupAction is generated, while a ConditionalAction is generated for a rule applied in try mode.

**Proposal 5.21.** The sequence of MT rule applications are defined by explicit ControlFlow restrictions set upon the sequence of corresponding rule actions.

As only succeed edges may lead from loop and forall nodes of a CFG such rule applications are modeled by GroupActions, which is simple a collection of subactions. However, in case of try rules, the CFG branches depending on the success of rule application, thus the corresponding action of a try rule must return whether the application of the rule was successful or not. After that, the composite actions of rules can be simple connected by ControlFlow objects in accordance with the CFG.

**Example 5.22.** The control flow graph of our reachability example of Fig. 4.10 is depicted in the AS notation in Fig. 5.15 stating that the execution of GroupAction initR should precede the execution of reachR action.

### 5.4.4 Action Semantics for pattern matching

The implementation of a graph pattern matching algorithm within Action Semantics is the central part of the entire encoding. The main challenge relies in the fact that graph transformation tools are traditionally control-oriented with global (constraint-based) graph pattern matching algorithms while

\(^1\) In order to avoid confusion with ASM rules, the term “rule” refers to model transformations rules if not explicitly stated otherwise in the following.
Action Semantics provides a data flow based specification technique allowing only local navigations for pattern matching algorithms.

In addition, the encoding of pattern matching depends on the rule application mode, thus, the same rule may have different AS representations when applied in different application modes. Loop and try modes are handled almost similarly (loop mode is based upon try mode, since a rule is tried to be applied once as long as possible and the next application depends on the applicability of the current one), and they differ essentially from the behavior of forall mode (where rule applications are executed in parallel for each matching).

In the sequel, we discuss the encoding of a rule applied in forall mode on the demonstrative example of Algorithm 18 for initR, while the pseudo encoding of reachR is shown in Algorithm 19 with detailed explanations given later in the current section. (Note that node identifiers in rules correspond to variables to ease the comparison of graph transformation rules with their pseudo AS representation.)

**Algorithm 18 Encoding initR in a pseudo action specification language**

1: GroupAction Automaton:initR() =
2: Variable a1, s1, S1;
3: a1 = ReadSelfAction();
4: if ReadIsClassifiedObjectAction(Automaton, a1) then
5:   {S1} = ReadLinkAction(a1, initial);
6:   for all s1 ∈ {S1} do {MapAction; parallel execution}
7:     if ReadIsClassifiedObjectAction(State, s1) then
8:       if ¬ testLink(a1, reachable, s1) then
9:         CreateLinkAction(a1, reachable, s1);
10:    end if
11:   end if
12: end for
13: end if

**Starting point of pattern matching**

The first step, which is to find the starting point for pattern matching is, in fact, identical for all modes.

**Proposal 5.23.** The starting point of the pattern matching is identified by the instance retrieved by a ReadSelfAction executed on an instance of the model class (i.e., Automaton in our example) and stored in a variable (see Line 3 in Algorithm 18).

As for the AS representation, a data flow is required to connect the output pin of ReadSelfAction with the input pin of AddVariableValueAction action. We must also specify that the previous value stored in the variable should be overwritten by setting the isReplaceAll variable to true. Note that matching instances of LHS graph nodes will be stored in AS variables (later on as well).

**Example 5.24.** The AS representation of Line 3 in Algorithm 18 is depicted in Fig. 5.16. We expect to retrieve an Automaton instance stored in variable a1.
Type checking of objects

Our next problem to be solved is to visit only type conforming objects when matching patterns.

**Proposal 5.25.** When an new object is obtained at any time during pattern matching (i.e., matched to a corresponding node in the graph transformation rule), we immediately test whether it has a conforming type (conformant to the type of the graph node). See Lines 4 and 7 in Algorithm 18 as examples.

This testing is performed by a ConditionalAction with a test clause consisting of a single ReadIsClassifiedObjectAction. The test action ReadIsClassifiedObjectAction (checking whether an object is an instance of a certain class) has to return a boolean value on its output pin, which serves as the test output for the test clause in the meanwhile. If value retrieved by the test subaction of a clause is true then the body action of the clause can be executed (which consists of further actions of pattern matching in our case).

**Example 5.26.** The AS representation of Line 7 in Algorithm 18 is depicted in Fig. 5.17. We check whether the value stored in variable s1 is an instance of the class Automaton. Naturally, we first have to get this value from the variable by a ReadVariableAction connected to the input pin of ReadIsClassifiedObjectAction by a data flow.

![Fig. 5.16. Starting point of pattern matching](image)

**Fig. 5.16. Starting point of pattern matching**

if ReadIsClassifiedObjectAction(s1, State) then ...

![Fig. 5.17. Checking types of objects](image)

**Fig. 5.17. Checking types of objects**
Navigating links

The core operation of graph pattern matching in a UML environment is the navigation of links.

Proposal 5.27. When a certain object is matched, the neighbors of the object (connected by links corresponding to the edge types in the graph transformation rule) are obtained by navigating links (Line 5 in Algorithm 18). This navigation results in a single object or a collection of objects stored in a variable.

A navigation of a link in AS (by applying ReadLinkAction) means that

- exactly one end object (called the source object) of a link is already known (i.e., at most one LinkEndData may have an associated single value on its input pin), while the target end of the link should be unknown prior to executing the action (naturally, an association can be navigated in both directions if allowed by the metamodel);
- the link should correspond to a certain association (also defined by the LinkEndData);
- as a result of the navigation a single object or a set of objects is retrieved (depending on multiplicities of the association and the topology of interconnected objects), and stored in a variable.

\[
(S1) = \text{ReadLinkAction}(a1, \text{initial})
\]

Fig. 5.18. Navigating links

Example 5.28. The AS representation of Line 5 in Algorithm 18 is depicted in Fig. 5.18. We read the value of variable \( a1 \) into the input pin of one LinkEndData corresponding to an association end of the initial association. When the ReadLinkAction is executed the result is written into variable \( S1 \) (variables with a capital initial will store a collection of objects in the sequel).
Rule application mode specific processing of collections

The processing of collections obtained from navigating links depends on rule application modes.

Proposal 5.29. When processing a collection for the pattern matching of a rule applied in \textit{forall} mode (see Line 6 in Algorithm 18), each element in the collection must be processed independently from each other, thus subsequent actions in the pattern matching process should be applied for each of them. For this reason, a MapAction is required in AS.

Algorithm 19 Encoding \textit{reachR} in a pseudo action specification language

\begin{verbatim}
1: GroupAction Automaton::reachR() =
2: Variable isApplicable, a1, s1, S1, s2, S2, t1, T1;
3: AddVariableValueAction(isApplicable, T);
4: while isApplicable do {LoopAction}
5: AddVariableValueAction(isApplicable, F);
6: a1 = ReadSelfAction();
7: if ReadIs Classified ObjectAction(Automaton, a1) then
8: {S1} = ReadLinkAction(a1, reachable);
9: for all s1 ∈ {S1} do {IterateAction, sequential execution}
10: if ¬isApplicable ∧ ReadIs Classified ObjectAction(State, s1) ∧
11: testLink(a1, states, s1) then
12: {T1} = ReadLinkAction(s1, source);
13: for all t1 ∈ {T1} do {IterateAction, sequential execution}
14: if ¬isApplicable ∧ ReadIs Classified ObjectAction(Transition, t1) ∧
15: testLink(a1, transitions, t1) then
16: {S2} = ReadLinkAction(t1, target);
17: for all s2 ∈ {S2} do {IterateAction, sequential execution}
18: if ¬isApplicable ∧ ReadIs Classified ObjectAction(State, s2) ∧
19: testLink(a1, reachable, s2) then
20: CreateLinkAction(a1, reachable, s2);
21: end if
22: end for
23: end if
24: end for
25: end if
26: end for
27: end if
28: end while
\end{verbatim}

Proposal 5.30. When processing a collection for the pattern matching of a rule applied in \textit{try} or \textit{loop} mode (see Lines 9, 12 and 15 in Algorithm 19), each element in the collection must be processed sequentially (thus an IterateAction is required in AS); however, the next element in the collection needs to be processed only if no complete matching has been found successfully so far (first boolean condition in Lines 10, 13 and 16).

Proposal 5.31. In both cases, the current element of the collection (corresponding to a matching of a certain node in the LHS of the rule) is stored in a variable. If the collection is empty then none of the subactions are executed thus this instantiation can be omitted.

Example 5.32. In the AS representation (see Fig. 5.19 for Line 6 in Algorithm 18), we first read the collection variable (ReadVariableAction) \textit{S1} into the input pin of MapAction with a further constraint stating the value contained by the subinput (output) pin should be stored in the variable \textit{s1} (AddVariableValueAction).
A weakness in the standard: Testing the existence of links

At the current point, we give explanation for Line 8 in Algorithm 18, which will demonstrate a major weakness in the AS standard.

When all the nodes of the LHS are instantiated by objects, we we have to test the existence of links between these objects prescribed by edges in the LHS that have not visited by navigation (or the non-existence of links in case of negative conditions). For efficiency reasons, such tests should be performed as soon as possible.

Example 5.33. For instance, in Line 8 of Algorithm 18, we check the negative condition that there must not be any existing reachable links between the automaton object stored in variable \( aI \) and the initial state object stored in \( sI \) by executing the boolean function \( \text{testLink}(aI, \text{reachable}, sI) \).

Problem 5.34. The existence of a link of a certain type relating two given objects cannot be tested within AS.

Function \( \text{testLink} \) can be used in UML models with AS expressions, but it, cannot be implemented by predefined actions of AS since a user function can be declared (with a signature specification) within AS as an action, but they cannot be defined (no specification of behavior). Typically, this action is implemented in every UML CASE tool with an action specification language; however, it is not part of the standard, which is very unfortunate.

Proposal 5.35. Thus, the standard should be extended with a \( \text{TestLinkAction} \) with two \( \text{LinkEndData} \) (both required to store an associated object on its input pin) as storing the input and a boolean valued output pin.

Without such an extension to the standard, the implementation of the external mathematical function a \( \text{testLink} \) will be CASE tool dependent and does not fit well to the generality of our approach.

5.4.5 Manipulating links and objects

Finally, after a successful pattern matching phase, objects and links are to be manipulated according to the difference of the LHS and the RHS of a rule.
Proposal 5.36. Whenever a transformation rule prescribes

- the *addition of an object*, a CreateObjectAction is executed and the created object is stored in a new variable.
- the *deletion of an object*, a DestroyObjectAction is executed on an object retrieved from the corresponding variable.
- the *addition of a link*, a CreateLinkAction is executed to create a link of a certain type between the objects read from the corresponding variables.
- the *deletion of a link*, a DestroyLinkAction is executed to destroy a link of a certain type between the objects read from the corresponding variables.

![Diagram of UML Action Semantics for Model Transformation Systems](image)

**Fig. 5.20.** Creating objects and links

Example 5.37. The AS encoding of link creation in Line 9 of Algorithm 18, and an fictitious creation of a State object is demonstrated in Fig. 5.20.

- When a new State object is created (upper part of Fig. 5.20) by CreateObjectAction it is directly passed to AddVariableAction by a data flow connection to store the new object in variable \( s1 \).
- When a reachable link is created between objects \( a1 \) (of type Automaton) and \( s1 \) (of type State), a CreateLinkAction is executed where values of both LinkEndCreationData are specified by data flows from the corresponding ReadVariableActions, while the ends of both LinkEndCreationData are defined by the related association in the metamodel.
5.4.6 Correctness of the encoding

Since the Action Semantics language consists of a large number of primitive actions, a full proof of correctness (i.e., to validate that the application of a model transformation rule is correctly implemented by the AS expression we derived) would exceed the limits of the current thesis.

However, since we followed a denotational approach (i.e., each graph transformation rule is compiled into a separate AS expression), therefore, it is sufficient to show that the current (specific) model transformation of the transformation designer is correct (see also meta-level vs. model-level proofs in Sec. 7.3.2). In the following, we demonstrate on the running example of the reachability problem how to construct such proofs, more specifically, we show that the AS representation of rule initR is correct.

In order to reason about the correctness of our encoding, we need to establish again a common semantic domain for model transformation systems and AS expressions. Unsurprisingly, ASMs are used again for this purpose.

Based upon the model-level ASM encoding of model transformation systems (presented in Sec. 4.3), we derive the following piece of ASM code for rule initR presented in Alg. 20.

Algorithm 20 ASM semantics of the graph transformation rule initR applied in forall mode

\begin{verbatim}
rule forall(initR) =
  1: forall A1, S1, C1 with
     automaton(A1) ∧ state(S1) ∧ initial(C1, A1, S1) do
  2:   create C2 do
  3:      addConnectionReachable(C2, A1, S1)
  4:     end create
  5: end for
\end{verbatim}

Based upon the ASM encoding of AS expressions of Table 5.2, Alg. 21 enlists the ASM representation of rule initR.

Algorithm 21 ASM semantics of the AS representation of initR

\begin{verbatim}
rule asInitR(self) =
  1: a1 := self
  2: if instanceOf(at, Automaton) then
  3:     S1 := \{a1 | entity(a1) ∧ entity(s1) ∧ ∃c1 : initial(c1, a1, s1)\}
  4:     forall s1 with s1 ∈ S1 do
  5:       if isXInstanceOfY(s1, State) then
  6:         if ∃c : reachable(c, a1, s1) then
  7:           create c2 do
  8:             addConnectionReachable(c2, a1, s1);
  9:           end create
 10:      end if
 11:   end if
 12: end for
 13: end if
\end{verbatim}

Proposition 5.38. Let \( \mathfrak{A} \), \( \mathfrak{B} \) and \( \mathfrak{C} \) be ASM states of the VPM representation of a finite automaton. Moreover, let \( r_{gd} \) denote forall(initR) while \( r_{as} \) denote asInitR.

- **Completeness.** For all \( \mathfrak{B} = \text{next}_{r_{gd}}(\mathfrak{A}) \), there exists an execution of the AS (with an automaton a passed as parameter to \( r_{as} \)) which yields the same state as result, i.e., \( \exists \mathfrak{C} : \mathfrak{C} = \text{next}_{r_{as}}(\mathfrak{A}) ∧ \mathfrak{B} = \mathfrak{C} \).

- **Correctness.** For all \( \mathfrak{B} = \text{next}_{r_{as}}(\mathfrak{A}) \) where \( r_{as} \) is called with an automaton a passed as parameter, there exists an execution of the GTS which yields the same state as result, formally, \( \exists \mathfrak{C} : \mathfrak{C} = \text{next}_{r_{gd}}(\mathfrak{A}) ∧ \mathfrak{B} = \mathfrak{C} \).

A similar proposition can be easily established for rule reachR.
5.5 Conclusions

In order to support the implementation of model transformations, I proposed automated program (and model) generation techniques that automatically synthesize a transformation program from the high-level specification of the transformation (defined by metamodeling and graph transformation techniques) which can be executed on an arbitrary model of the modeling language(s).

1. Automated program generation by model transformation. I proposed a reflective way (consisting of consecutive model transformation steps) to provide automated program generation for model transformations (Sec. 5.2.3 based on [167,178]).

2. A Prolog implementation of model transformation virtual machine. I elaborated a Prolog implementation of a virtual machine executing high-level specifications of model transformations (Sec. 5.2.1 and Sec. 5.2.2, respectively, following [166,167,178]). The implementation of graph transformation rules is based upon a graph pattern matching algorithm using depth-first search with optimized clause ordering and the powerful unification mechanism of Prolog. The realization of the control flow graph exploits the meta-programming facilities of Prolog.

3. Action Semantics description of model transformations. I proposed how to map model transformations into standard Action Semantics expressions (Sec. 5.4 following [173,185]). Graph transformation rules are simulated by an algorithm using local searches, while the control flow graph is implemented by built-in compound actions of the Action Semantics standard.

Conceptual relevance

The main advantage of the Prolog approach (already implemented in the VIATRA framework) is its high-level and reflective nature. In this sense, only a functional core of VIATRA was written by hand while the implementation of other parts were generated automatically from a high-level specification. This is similar to the bootstrapping process of compiler design, where, for instance, a C compiler is written in C and compiled by an existing C compiler, and recompiled by itself afterwards to provide a more efficient and reliable target code.

Practical relevance

As a main practical relevance of the chapter, the program generation methodology based on model transformations (of Sec. 5.2.3) may also serve as the underlying semantic and implementation basis of future model-driven code generators yielding provenly correct application code. Instead of the currently dominant hard-wired code generators, the designers may modify code generators using a very-high level specification language. Also the concept of raising the code up to a model-level fits better to the MDA philosophy.

The main practical benefit of the AS approach is that visual but mathematically precise model transformations can be directly encoded in the standard action specification language of the MDA (and UML) environment (having extensive tool support such as BridgePoint [139] or xUML [95]) thus providing a smooth integration of formal specifications and industrial standards. The entire encoding was demonstrated on a small example (reachability analysis of finite automata) which was still rich enough to cover all the basic rules of our encoding.

As a summary, while efficient implementations of graph transformation engines (see the conclusions of Chapter 4) may serve as (meta-)interpreters for various modeling languages, the automated program of the current chapter should support to compile and integrate these high-level specifications into for various industrial platforms.

Limitations

During benchmark application in a research project [91], we first experienced a performance degradation in our transformation-based code generation approach. After thorough investigations, we realized
that the final model-to-code step of the model transformation-based program generation methodology can be inefficient for complex transformations. Recent research has aimed to combine pattern-based and graph transformation-based code generation techniques [158] to manage this performance problem.

Fortunately, the Prolog engine itself that carries out the model transformation was not a performance bottleneck of the transformation based verification and validation framework. In fact, the run-time of a model transformation was typically just a few percentage of the subsequent analysis phase (using a model checker, for instance). Increased run-time was also experienced in [132] where the VIATRA model transformation engine was also used as a mathematical analysis framework (and not just for model transformations).

Future work

Future work should primarily aim at better exploiting the connection between attribute grammars (widely used in compiler construction to define transformations between textual languages) and graph transformations. A synergy between these paradigm may help in providing a precise foundation and probably more efficient implementation of both code-to-model (reverse engineering) and model-to-code transformations (code generation).

In the upcoming chapter, we cope with the automated formal verification of modeling languages using model checking techniques. Therefore graph transformation rules will be projected into transition systems, which provide the input language of many model checking tools.
Automated Formal Verification of Visual Modeling Languages by Model Checking

I present a meta-level transformation technique to enable model checking-based symbolic verification for arbitrary well-formed models and modeling languages (with formal semantics defined by graph transformation systems) by projecting them into state transitions systems that serve as the underlying mathematical specification formalism of various model checker tools. The feasibility of our approach is demonstrated by modeling and analyzing a well-known verification benchmark both on the model and metamodel level.

6.1 Towards Model Checking Graph Transformation Systems

For many years, the abstract syntax of UML (including OCL [124], Action Semantics [123] and related domain specific profiles) has been defined visually by means of metamodeling. As discussed in Chapter 2 a straightforward representation of such models and languages can rely on the use of directed, typed, and attributed graphs as the underlying semantic domain.

In this sense, graph transformation [142] has recently become very popular as being a general, rule-based visual specification paradigm to formally capture (i) requirements, constraints and behavior of UML-based system models [34,62], and (ii) the operational semantics of modeling languages based on metamodeling techniques [53,61,101,155,167,168,181]. Similar ideas are applied directly on formalizing transformations from UML into various semantic domains (Petri nets, SOS rules, dataflow nets, etc.) [63,191].

Problem statement

While graph transformation is very popular as a high-level and expressive specification formalism, the lack of proper techniques and, especially, tools for the formal analysis of such specifications aiming to decide whether a certain user requirement (such as the absence of deadlocks, safety and liveness properties) holds in the system model hinders the use of graph transformation systems (GTS) in an effective design process for systems and visual modeling languages.

While several conceptual approaches have been proposed recently as a formal analysis technique for GTSs, existing graph transformation tools only provide simulation capabilities to assess whether a certain requirement holds in the system model, which is frequently insufficient for verification purposes. Since developing an analysis tool from scratch is very costly, one should possibly exploit existing model checker tools to carry out the verification of GTSs.

Related work

Unfortunately, the formal verification of models (and modeling languages) defined by graph transformation systems has remained so far on a rather theoretical (conceptual) level.
• The theoretical basics of verifying open graph transformation systems by model checking techniques have already been studied thoroughly in, e.g., [82,83] (and subsequent papers). The authors propose that graphs can be interpreted as states and rule applications as transitions in a transition system. Unfortunately, as we demonstrate in the paper, this direct encoding of graphs into a model checking problem is, unfortunately, infeasible in practice since model checkers will easily run out of space due to this verbose state representation.

• Ongoing research into the same direction (with an envisaged tool support) has recently been sketched in [140] aiming to extend earlier results on reasoning about allocation and deallocation problems [56].

• A recent framework [12,13] aims at analyzing a special class of hypergraph rewriting systems by a static analysis technique (based on foldings and unfoldings of a special class of Petri nets). This framework is able to handle infinite state systems by calculating a representative finite complete prefix. However, the class of GTSs they handle has certain drawbacks from a practical, model engineering point of view concerning intuitiveness and expressiveness. Probably the most severe of these limitations is that the removal of nodes is not allowed. In contrast, graph transformation rules in our approach are arbitrary rules following the single pushout (SPO) approach [59] (with straightforward extensions to the double pushout (DPO) approach [45]); however, the price we have to pay is that our graph transformation system has to be a priori bounded.

• Rule invariants have been proposed lately in [129] in analogy with the notions of transition invariants in Petri nets. The authors also transfer the traditional concepts of liveness, boundedness, etc. to GTSs. The main conceptual limitation from a verification point of view is that rule invariants in a GTS (like transition invariants in a Petri net) only provide a semi-decision technique. In other words, they detect potential cycles in the system, some of which might never occur on any execution paths. On the other hand, rule invariants can be computed efficiently only from the static structure of the GTS.

As a summary, none of the frameworks give direct suggestions on concrete implementation or tool support how to verify formal specifications given in the form of graph transformation systems by existing model checking tools.

6.1.1 Objectives

In the current chapter, we extend our initial ideas already discussed in and propose a meta-level and optimized technique [171,172] with benchmark examples to verify graph transformation systems used as either a model-level or meta-level formal specification technique by existing model checkers.

• Graph transformation on the model-level. For any user model with structural descriptions in the form of traditional class and object diagrams and dynamic behavior captured by graph transformation systems (like FUJABA [114] or PROGRES [151] models as practical examples), we project it into a behaviorally equivalent transition system (TS).

• Graph transformation on the meta/language level. For any well-formed model of any high-level modeling language (with abstract syntax defined by metamodeling and operational semantics formalized by graph transformation), we generate a separate (metamodel and model-specific) transition system that faithfully represents the behavior of the model instance.

We present a meta-level analysis technique where only the semantics of a modeling language should be defined precisely when a new modeling language is constructed, and then the formal analysis can be carried out automatically (without designing individual mappings into analysis tools). In addition, the maintenance of meta-level analysis frameworks (like the one proposed in the current paper) is much easier since modifying the semantics of language on the GTS level is less erroneous then modifying a complex translation program.
The output TS is generated in two steps. First all potential applications of a graph transformation rule are collected into separate transitions by a Cartesian product construction. Then, in a second phase, an optimization is carried out which eliminates the static parts and dead transitions from the target TS to drastically reduce the state space.

In order to capture the translation problem at the right level of abstraction, moreover, to gain a certain level of independence of particular model checker tools, both the source and the target model of our mapping (and the mapping itself) will be defined in the form of abstract state machines [74] (i.e., both the source GTS and the target TS). This uniform semantic representation highly eases to formalize and prove the correctness and completeness of our approach.

The input languages of concrete tools can be generated by further preprocessing step (which is rather syntactic and thus out of the scope of the current paper). Meanwhile, running examples on model checking specifications will always be given in the concrete tool format of the SAL framework [23] to provide guidelines on how to tailor our technique to existing tools.

The practical feasibility of our approach will be demonstrated on the well-known example of the dining philosophers, which is a common benchmark for assessing the performance of verification tools. The dining philosophers’ problem will be modeled and analyzed in different ways (with graph transformation appearing both on the model-level and meta-level) to prove deadlock freedom and safety properties.

As a summary, the main benefits of our approach are the following.

1. We present a meta-level analysis techniques which is parameterized by (the metamodel of) the modeling language (Sec. 6.3.2).
2. We present an optimization technique that reduces the number of state variables and eliminates dead transitions to avoid state space explosion (Sec. 6.3.3).
3. We build on existing model checker tools (such as SAL [23] or Murφ [2]) to speed up the verification process (and the work related to implement the verification tool itself).
4. We assess the practical feasibility of our approach on a verification benchmark (Sec. 6.5).

As an overview, we present the information flow of the chapter in Fig. 6.1.

Finally, we also explicitly summarize the limitations/prerequisites of our approach (although the detailed explanations of these limitations will be provided later on in the current chapter).

1. We assume that the structure of a modeling language is defined by a metamodel (UML class diagram), while the dynamic behavior is captured operationally by graph transformation rules.
2. We assume that an initial instance model (UML object diagram) is also provided by the user.
3. We suppose the links (edges) are relations on objects (nodes) thus they do not have identities.
4. We assume that attributes of objects have finite domains.
5. We assume that an explicit upper bound is a priori known to each class in the metamodel (e.g., a class A is allowed to have at most 5 instances in an instance model).

Note that while our technique is applicable to modeling languages from arbitrary domains (if defined by means of metamodeling and graph transformation), in the paper, we will rather focus on applying it for software engineering purposes, as proving that an IT system will not collapse under within conditions is probably the most challenging task in our opinion.

6.2 Model Checking Transition Systems

6.2.1 Transition systems

Transition systems (TS) (pp. 71-75 in [135]) are a common mathematical formalism that serves as the input specification of various model checker tools. They have certain commonalities (in many cases on the concrete language level as well) with structured programming languages (like C or Pascal) as the system/program is evolving by executing non-deterministic if-then-else like rules that manipulate state variables. In all practical cases, we must restrict the state variables to have finite domains, since model checkers typically traverse the entire state space of the system to decide whether a certain property is satisfied.

Definition 6.1. Formally, a transition system is a 4-tuple $TS = (V, Dom, T, Init)$ where

1. $Dom = \{D_1, \ldots, D_m\}$ is a set of finite domains
2. $V = \{v_1, \ldots, v_k\}$ is the set of state variables taking their values from a corresponding domain. The domain of a variable is denoted as $dom(v_j) = D_j$ or shortly as $v_j : D_j$ where $D_j \in Dom$;
3. $T = \{\tau_1, \ldots, \tau_n\}$ is the set of transitions (guarded commands) which is of the form $p \rightarrow v'_1 := e_1, \ldots, v'_n := e_n$ where the $p$ is a boolean guard condition, and an action (or assignment) $v'_j := e_j$ specifies an update for state variable $v_j$;
4. $Init$ is a (unquantified first order) predicate defining the initial state.

Similarly to the majority of model checker tools, we suppose that state variables can be stored in state variable arrays $p_1, \ldots, p_m$ ranging on (sets of) object identifiers, and they can be referred to as $p_j[i] = v_i$. In this case, an $n$-dimensional state variable array $p[i_1] \ldots [i_n]$ has $n$ index domains $ID_j \in Dom$ (for which $i_j \in ID_j$), and a value domain $VD \in Dom$. The intended meaning is that each state variable $v_{i_1, \ldots, i_n} = p[i_1] \ldots [i_n]$ (i.e., a location in the state variable array) takes its value from the value domain $VD$. Naturally, all index and value domains are required to be a priori finite. For instance, we will declare a one-dimensional state variable array color later in Example 6.4 with an index domain $StateID = \{s1, s2, s3\}$ and value domain $ColorType = \{R, G, B\}$.

Note that transition systems are a kind of an abstract syntax of a specification language, which can be used to generate a state space (formally, a Kripke structure) describing the system.

Definition 6.2. A Kripke structure $KS = (\Sigma, N, I, \sigma)$ is a four tuple where (i) $\Sigma$ is the set of states (induced by all possible evaluations of state variables), i.e., $\Sigma = \{dom(v_1) \times \ldots \times dom(v_k)\}$; (ii) $N \subseteq \Sigma \times \Sigma$ is the transition relation defined as $N = \bigcup_{i=1}^{n} Act_{\tau_i}$ where $Act_{\tau_i}$ is a relation induced by the guarded commands of the TS, i.e., $Act_{\tau_i}(V, V') = p_i \land \bigwedge_{i=1}^{k} v'_{i,j} = e_{i,j}$; (iii) $I \subseteq \Sigma$ is the set of initial states; and (iv) $\sigma : \Sigma \rightarrow 2^AP$ is a labeling function mapping each state to a subset of atomic propositions $AP$ (e.g., atomic equations) that are valid in the given state.

Intuitively, a transition (guarded command) of the form $p \rightarrow v'_1 := e_1, \ldots, v'_n := e_n$ can be executed in any state satisfying condition $p$. Thus condition $p$ is called the enabledness (guard) condition of the transition $\tau$ and denoted as $en_{\tau}$. We say that $\tau$ is enabled in a state $s$, if its condition $p$ is satisfied
in $s$ (denoted as $s \models^K S p$ or simply $s \models p$). The effect of executing $\tau$ is that (i) all expressions $e_1, \ldots, e_n$ are calculated first based upon $s$, and then these new values are assigned to state variables $v_1, \ldots, v_n$.

An execution path of a Kripke structure is an infinite sequence of states $s_0, s_1, s_2, \ldots$ starting from one of the initial states ($s_0 \models Init$) and progresses from one state to another by non-deterministically selecting and firing (enabled) transitions of the system.

The requirements (or properties to be verified) for models specified by a Kripke structure are frequently captured by some temporal logic formulae. However, since only safety properties and deadlock freedom are being proved in the current paper, we define these concepts without the use of temporal logic operations. As our technique focuses on the transformation of the system specification (which is independent of expressing requirements), we suppose that the requirements can be expressed by some formalism understood by the target model checker.

- **Safety properties.** A safety property $\phi^s$ (pp. 171–172 in [135]) is an invariant (a boolean expression composed of atomic predicates) that must hold in each state of the system. Whenever a state is reached during the traversal of the state space where this property is violated, model checking can terminate immediately with an error message.

- **Deadlock freedom.** A system is in a deadlock (pp. 72–73 in [135]), if no transitions are enabled at the specific state. Here the corresponding deadlock property $\phi^d$ could be derived by combining the guards of transitions.

Now a simplified definition of the traditional model checking problem (for handling safety and deadlock properties) is as follows.

**Definition 6.3 (Model checking problem).** Given (i) a system model in the form of a transition system $TS$ (inducing a Kripke structure $KS$), and (ii) a safety property $\phi$, then the model checking problem can be defined as to decide whether $\phi$ holds on each execution path of the system (i.e., whether $s_i \models \phi$ for all $s_i$ on the execution path). Moreover, the system should be free of deadlocks, i.e., $\forall i : \exists \tau \in T : s_i \models \text{en}_{\tau}$.

After the mathematical definitions, we overview the concepts of a specific model checker tool that will provide the notation for examples on transition systems, since the language itself is very close to the mathematical definition.

6.2.2 SAL: Symbolic Analysis Laboratory

The SAL (Symbolic Analysis Laboratory) [23] framework aims at combining different tools for abstraction, program analysis, theorem proving, and model checking for the evaluation of system properties. The SAL architecture can be interpreted as a “tool-bus” where a collection of tools interact through the common intermediate language of transition systems. The individual analyzers (theorem provers, model checkers, static analyzers) are driven from this intermediate layer and the analysis results are fed back to this intermediate level.

- In the SAL intermediate language, the unit of specification is a context, which contains declaration of types, constants, transition system modules, and assertions. A basic SAL module is a state transition system where the state consists of input, output, local, and global variables, which refer to different access modes.

- A basic module also specifies the initialization and transition steps. These can be given by a combination of definitions or guarded commands. A definition (or assignment) is of the form $x = \text{expression}$ or $x' = \text{expression}$, where $x'$ refers to the new value of variable $x$ in a transition. A guarded command is of the form $g \longrightarrow S$, where $g$ is a boolean guard and $S$ is a list of definitions of the form $x' = \text{expression}$. In addition to that, we may also define (auxiliary) functions that always yield a deterministic result.
SAL modules can be composed (i) synchronously, so that $M_1 \parallel M_2$ is a module that takes $M_1$ and $M_2$ transitions in a lockstep, or (ii) asynchronously, when $M_1 \parallel M_2$ is a module that takes an interleaving of $M_1$ and $M_2$ transitions.

Example 6.4. The SAL example below defines two domains (StateID and ColorType) and a one-dimensional state variable array color mapping (elements of) StateID to ColorType. For initialization, we assign the values $R$, $G$, and $B$ to the locations $s1$, $s2$, $s3$ of the array, respectively. The single transition (guarded command) states that values $R$ and $G$ can be swapped at locations $s1$ and $s2$ (respectively) of array color.

% Domains
StateID : TYPE = {s1, s2, s3};
ColorType : TYPE = {R, G, B};
% State variables
GLOBAL color: ARRAY StateID OF ColorType
% Initialization predicate
INITIALIZATION
color[s1] = "R"; color[s2] = "G"; color[s3] = "B";
% Guarded commands
TRANSITION
color'[s1] = "R" AND color'[s2] = "G" ->
color'[s1] = "G"; color'[s2] = "R"

In the paper, we will use the SAL specification language for code level examples when describing graph transformation systems as traditional state transition systems despite the fact that the SAL framework is not yet available for public use. However, as SAL is aimed to provide a general front-end to many individual model checkers, we can achieve a high level of independence from concrete tools in exchange.

6.2.3 An ASM encoding of transition systems

Since abstract state machines are generalizations of transition systems, the ASM encoding of transition systems is rather straightforward.

State variables, Domains, Initialization

Since state variables are arranged into state variable arrays, we may assign a function symbol $p$ for each state variable array $p[x]$. As a special case, a nullary dynamic function symbol in an ASM may represent a single state variable in a TS.

In TSs, each index and value domains $D_i$ should be a priori bounded. Therefore we may assign sorts to function parameters and return values to express the fact that each parameter should be taken from the corresponding domain. As a result, for a TS with an $n$ dimensional state variable array $p$ with corresponding index domains $ID_i$ and value domain $VD$, in the ASM representation we have $p(x_1 : ID_1, \ldots, x_n : ID_n) : VD$.

Thereafter the initialization predicate simply defines the initial state $x_{init}^1$ in the ASM representation of the TS. Since, in general, the initial state of a TS may be non-deterministic, the ASM representation of the initial state may depend on the environment (in other terms, it may contain free variables).

Transitions (guarded commands)

The most crucial restriction of TSs (in contrast to ASMs) is that non-determinism is only allowed to select from a set of enabled transitions, but if someone already selected a single transition $g \rightarrow p_1[v_1] := e_1, \ldots, p_n[v_n] := e_n$, it should be deterministic. Therefore the guard $g$ and the expressions $e_i$ must not contain free variables, and the use of the choose construct is also prohibited in the ASM.
6.3 From Graph Transformation Systems to Transition Systems

In the current section, we provide a meta-level approach to map graph transformation systems into transition systems in order to verify properties of user models by model checking tools.

In other words, we propose a translation that inputs (i) the metamodel of a visual modeling language (or class diagram, on the model level), (ii) its operational semantics (dynamic behavior) in the form of a graph transformation system, and (iii) a concrete, well-formed model instance of the language (object model), and generates a transition system as the output.

Note that since we use model checkers, we do not reason about the properties of the language itself (as done by theorem provers). However, we can automatically prove certain correctness properties (like safety and deadlock freedom in our case study) for a well-formed specific but arbitrary instance model of the language.

It is essential to be pointed out that in practical cases, the user is only interested in the correctness of his or her model and not the correctness of the entire modeling language. Moreover, proving the correctness of a property for all valid model instances is often impossible.

### 6.3.1 A conceptual overview of the encoding

As demonstrated previously by [82,83], graph transformation systems can be interpreted as a TS where the state space is constituted from attributed graphs created by elementary graph transformation steps (see Fig. 6.2 for an overview).

![Fig. 6.2. The state space of graph transformation systems](image)

This state space has a special structure: while the graph representation of a user model is typically finite (for instance, infinite UML models are somewhat rare), attributes may result in potentially
infinite state representations (e.g., in case of integers or reals). As current model checking tools can only traverse state spaces induced by state variables of finite domain, variables having infinite domains should be abstracted to boolean domains before model checking, for instance, by a technique called predicate abstraction [145].

Thereafter, models having attributes of finite domain (either originally or after predicate abstraction) will form the states of the TS, and they will be encoded as predicates over node identifiers. Applying a graph transformation rule for a single match will be represented as a transition in the TS.

The major challenge in such an encoding is that while graph transformation is a meta-level specification technique, transitions in a TS are defined on the model level. As a consequence, a single graph transformation rule is encoded into several transitions in the TS; moreover, the same graph transformation rule may yield different enabled transitions (even during the same execution) when applied to different instance graphs.

### 6.3.2 A naive encoding of graph transformation systems into transition systems

First we present a naive encoding of GTSs into TSs, which easily demonstrates how to derive a set of transitions for a single graph transformation rule, but which is inefficient from a verification point of view.

**Mapping graphs into state variable arrays**

Since we introduced the same semantic representation for the graphs and state variable arrays, we can naturally map (i) each class into a one-dimensional boolean state variable array, (ii) each association into a two-dimensional boolean state variable array, and (iii) each attribute into a one-dimensional state variable array with enumeration range. However, since TSs are not as expressive as GTSs, the following additional assumptions are required.

- In order to define the corresponding index domains for these state variable arrays, we have to assume that there exists an a priori upper bound for the number of objects in the model for each class. In this respect, we suppose that when a new object is created it is only activated from the bounded “pool” of currently passive objects (deletion means passivation, naturally), and the same applies to the interpretation of links.
- Concerning the value domains of state variable arrays, attributes (slots) of infinite type have been abstracted into some representative finite domain, which is carried out by the user (potentially with tool support) using predicate abstraction, for instance.

The main limitation imposed by these restrictions is that we cannot handle graph transformation systems with potentially infinite state space. In other terms, we carry out bounded model checking, which is only able to efficiently traverse the state space up to a certain depth provided as a parameter. In our case, this parameter is the list of upper bounds for each class\(^1\). This list of upper bounds serves as an additional input to our translation (because an algorithmic “guess” for the upper bounds has computational complexity problems).

As a result of this encoding, we can trivially establish a mapping \( F_1 \) from the ASM states \( \mathcal{A}^{gr} \) of a GTS to the ASM state \( \mathcal{A}^{nv} \) of a naive TS since the same function symbols are used with the same interpretation. Formally, \( \mathcal{A}^{nv} = F_1(\mathcal{A}^{gr}) \) with (i) \( F_1(p) := p \) for all function symbol \( p \), and (ii) \( \overline{\phi(x = v)}^{\mathcal{A}^{nv}} := \overline{\phi(x = v)}^{\mathcal{A}^{gr}} \).

In this sense, the initialization predicate of the corresponding TS is thus defined by the initial instance model of the GTS.

\(^1\) The list of upper bounds is only required for dynamic classes, since we can easily calculate the upper bound for a static class depending on the model instance
Example 6.5. The naive SAL encoding of our finite automaton model (Fig. 2.2 and Fig. 2.3) would include the following lines.

\[\text{% Domains}\]
\[\text{AutID : TYPE = \{a1\};}\]
\[\text{StateID : TYPE = \{s1, s2, s3\};}\]
\[\text{fail : MODULE =}\]
\[\text{% State variable arrays}\]
\[\text{BEGIN}\]
\[\text{GLOBAL reachable: ARRAY AutID OF}\]
\[\text{ARRAY StateID OF BOOLEAN}\]
\[\text{GLOBAL states: ARRAY AutID OF}\]
\[\text{ARRAY StateID OF BOOLEAN}\]
\[\text{% Initialization predicate}\]
\[\text{INITIALIZATION}\]
\[\text{states[a1][s1] = TRUE;}\]
\[\text{states[a1][s2] = TRUE;}\]
\[\text{states[a1][s3] = TRUE;}\]
\[\text{reachable[a1][s1] = FALSE;}\]
\[\text{reachable[a1][s2] = FALSE;}\]
\[\text{reachable[a1][s3] = FALSE;}\]

Mapping graph transformation steps to transitions

The main task in encoding transformation steps (potential applications of graph transformation rules) into transitions of TSs is to simulate all the different behaviors imposed by the graph pattern matching process in a low-level structure. As graph transformation is a meta-level specification technique, a single graph transformation rule will be encoded into several transitions. In fact, all potential application of a rule (occurrences of a pattern) have to be collected at compile time and then enumerated explicitly as different guarded commands.

The formal ASM explanation for this “unfolding” of GT rules stems from the fact that the ASM representation of a GT rule may contain variables bound by a choose construct in its precondition formula, while such non-determinism is not allowed in the guard of a TS transition.

The number of potential transitions (according to a naive first estimation) is determined by the complexity of the LHS of a rule (i.e., the number of nodes), and the size of the model (i.e., the cardinality of domains of state variable arrays). We have to instantiate the variables of the LHS in all possible combinations by a Cartesian product construction to enumerate all potential matchings of the rule. Thus a node in the LHS is tried to be matched to each object in the model having a conformant type.

Note that the Cartesian product of variable instantiation should also enumerate all possible combinations of object identifiers taken from the reserve (i.e., for objects that are not present in the initial model). However, as we assume to have an a priori upper bound for the model, it is no longer a problem.

The process of generating a single transition can be divided into three phases, namely, the generation of (i) the positive guard corresponding to the LHS, (ii) the negative guard corresponding to NEG, and (iii) the action part.

- **Guard of the LHS.** For a specific potential matching (i.e., combination of variable instantiations) of the LHS, the guard of the LHS is derived straightforwardly from the ground LHS formula obtained after substituting the values to variables.
- **Guard of the negative condition.** In order to ensure that none of the potential instantiations of the variables of the NEG graph can be satisfied for the given potential matching, we have to generate a conjunction of negative clauses, where each clause is a possible instantiation of the variables in NEG (alternatively speaking, a potential matching of the NEG graph). The guard of the transition generated for the negative application conditions is obtained as before from the ground NEG formulæ.
• **Actions.** Prior to generating the actions that update certain locations in state variable arrays, we have to instantiate the variables of objects created by the GT rule application. In the paper, we abstract from this problem by assuming that there exists a special procedure `nextIdFromReserve()`, which retrieves the next location from the reserve at compile time. When all the variables in LHS and RHS are instantiated, we may simply copy the updates (in the ASM representation) of the GT rule.

The process of generating transitions in the TS from the ASM representation of a graph transformation rule is formally defined in a pseudo ASM code in Alg. 22.

### Algorithm 22: A naive generation of transitions in a TS

**Notation:**
- $\overline{X}_{lhs}$: variables related to nodes appearing in the Lhs;
- $\overline{X}_{neg}$: variables appearing only in the Neg but not in the Lhs (i.e., related to nodes in $Neg \setminus Lhs$);
- $\overline{X}_{val}$: variables appearing in Lhs or Neg, i.e., $\overline{X}_{lhs} \cup \overline{X}_{neg}$;
- $\overline{X}_{adl}$: variables related to nodes in $LHS \setminus RHS$ (where $\overline{X}_{adl} \subseteq \overline{X}_{lhs}$);
- $\overline{X}_{val}$: variables related to nodes in $RHS \setminus LHS$;
- $\overline{X}$: variables appearing in Lhs or RHS, i.e., $\overline{X}_{lhs} \cup \overline{X}_{val}$.

- **cl**: a predicate derived from a class
- **asc**: a predicate derived from an association
- **att**: a predicate derived from an attribute
- **parent ⇔ child**: parent is a superclass of child
- **undef**: the undefined value
- **val**: a value from a corresponding value domain

- **ζ**: variable assignment
- $x \mapsto a$: value $a$ is now assigned to variable $x$
- $dom(x)$: domain (or sort) of variable $x$

- $\phi_{lhs}$: the formula derived from the LHS of a rule
- $\phi_{neg}$: the formula derived from the Neg graph of a rule
- $p_{i,j}$: a predicate in the LHS formula
- $p_{i,j}$: a predicate in the Neg formula

`nextIdFromReserve()`: a special function to handle symmetries when creating object; called at compile time

$$\tau \equiv \mathcal{G} \rightarrow \mathcal{A}$$: a transition in the target TS with guard $\mathcal{G}$ and action $\mathcal{A}$

```plaintext
fun naive-transitions =
1: let $\overline{X}_{lhs} \equiv \{x_1, \ldots, x_n\}$, $\overline{X}_{pre} \equiv \{y_1, \ldots, y_m\}$,
2: forall $(a_1, \ldots, a_n) \in dom(x_1) \times \ldots \times dom(x_n)$ do
3: $\mathcal{G}_{lhs} := \{x_i \leftrightarrow a_i, \ldots, x_n \leftrightarrow a_n\}$
4: $\phi_{lhs} := \phi_{lhs}(a_1, \ldots, a_n) \equiv \bigwedge_i p_i$
5: $\phi_{neg} := \bigwedge_{x \in \overline{X}_{pre}} \neg \phi_{neg}(b_1, \ldots, b_m)$ where $(b_1, \ldots, b_m) \in dom(y_1) \times \ldots \times dom(y_m)$ and $\phi_{neg} = \bigwedge_{x \in \overline{X}_{pre}}$ $p_{i,j}$
6: forall $z_i$ with $z_i \in \overline{X}_{adl}$ do
7: $z_i \mapsto nextIdFromReserve(dom(z_i))$;
8: end for
9: forall updates $p_i(x_i) := v_i$ in rule $r_{xpo}$ do
10: $\mathcal{A} := par_i (p_i[x_i] := v_i)$
11: end for
12: $\tau := (\phi_{lhs} \land \phi_{neg}) \rightarrow \mathcal{A}$
13: generate $\tau$
14: end for
```

1. First we collect all variables $\overline{X}_{lhs}$ in the LHS and $\overline{X}_{pre}$ in the NEG graph of the rule (Line 1).
2. Then we process one by one all elements $(a_1, \ldots, a_n)$ in the Cartesian product of variable domains of $\overline{X}_{lhs}$.
a) In Line 3, we instantiate the variables \( x_1, \ldots, x_n \) with \( \langle a_1, \ldots, a_n \rangle \) to constitute a potential matching of the rule.

b) In Line 4, we substitute them into the LHS formula \( \phi_{lh_s}(\overline{x}_{lh_s}) \) to obtain the first part \( (g_{lh_s}) \) of the TS guard.

c) In Line 5, we proceed similarly to create the conjunction of the negative condition \( \text{NEG formula}(c) \phi_{neg}(\overline{x}_{neg}) \) to obtain \( g_{neg} \). This conjunction expresses that no assignments for variables \( \overline{x}_{prec} \) may satisfy any of the negative condition formulae \( \phi_{neg}(\overline{x}_{prec}) \).

d) We instantiate (in Lines 6–8) all variables in \( \overline{x}_{add} \) with elements taken from the reserve of the corresponding domains.

e) We define (in Lines 9–11) a TS action \( \text{act} \) as the parallel composition of updates.

f) Finally (in Lines 12–13), we generate a TS transition \( \tau \) composed \( g_{lh_s} \) and \( g_{neg} \) as guards and \( \text{act} \) as action.

Example 6.6. In case of rule \( \text{initR} \) (of Fig. 4.10), the corresponding SAL specification in a naive encoding is as follows.

```
TRANSITION % guarded commands for initR
% first potential match
automaton(a1) AND init(a1,s1) AND state(s1) AND
NOT (reachable[a1][s1]) --> % guard
reachable[a1][s1] = TRUE; % assignment

% asynchronous composition
% second potential match
automaton(a1) AND init(a1,s2) AND state(s2) AND
NOT (reachable[a1][s2]) -->
reachable[a1][s2] = TRUE;

% third potential match
automaton(a1) AND init(a1,s3) AND state(s3) AND
NOT (reachable[a1][s3]) -->
reachable[a1][s3] = TRUE;
```

Handling symmetries in object creation

Probably, one of the most crucial design decisions one has to make is concerned with the creation (activation) of objects in order to handle symmetries (or isomorphism) in an appropriate way. The problem relies in the fact that graph transformation only prescribes to introduce new objects (with fresh identifiers), while in a TS, we have to specify precisely which is the identifier for the new object.

A naive approach may simply generate all the different ways how a new object can be created (by testing whether an object having a certain identifier is active or not); however, such a solution would result in an unacceptable amount of transitions considering isomorphic (or symmetric) cases differently.

Our proposal (that was implemented in [147]) is to maintain a counter for each class of objects pointing to the next free object identifier. In this sense, a total order is defined on identifiers (starting, for instance, with initially active object identifiers), and when a new object is to be activated, the location indexed by the next identifier (according to the ordering relation) will be set to true.

The drawback of the solution is that each object is allowed to be activated only once in its lifetime (a cyclic activation and passivation is thus not allowed for the same identifier), which might require a larger input model to be considered. Since allowing to create each object once is close to the object-oriented philosophy, this limitation is not crucial in our opinion.

6.3.3 Optimizations in transition systems

When the current approach was applied for encoding and verifying UML statecharts formalized by graph transformation systems (following [168]), we revealed that the previous encoding consumes an
unacceptable amount of space when model checking even small applications. For instance, the encoding of an automaton having 20 states and 20 transitions requires more than 500 boolean state variables, which is typically far too many to be handled by state-of-the-art model checking tools (resulting in a state space having $2^{500}$ states).

The problem originates from the fact that in the previous naïve approach, both state variables and transitions were introduced “verbosely” for the static parts of a model as well.

**Eliminating static state variables**

Supposing that the structure of a finite automaton remains unchanged during the lifetime of the model, our translation needs to create state variables only for dynamic elements (such as current or reachable links in the metamodeled of finite automata), while static parts can be omitted by compile-time preprocessing.

**Example 6.7.** The optimized SAL encoding of our finite automaton model (Fig. 2.3) would include the following lines. Note that lines of code in Example 6.5 that are not part of Example 6.7 (such as the states state variable array) are eliminated as the result of our optimization.

```sal
AutID : TYPE = {a1};
StateID : TYPE = {s1, s2, s3};
fa1 : MODULE =
BEGIN
GLOBAL reachable: ARRAY AutID OF
  ARRAY StateID OF BOOLEAN
INITIALIZATION
  reachable[a1][s1] = FALSE;
  reachable[a1][s2] = FALSE;
  reachable[a1][s3] = FALSE;
```

Naturally, as a specific transition may never be applied during the execution of a specific model, such an optimized encoding may still introduce state variables for model elements that are never changed. However, the only possibility to eliminate such unreachable model parts requires at compile time the use of some sophisticated static analysis techniques on graph transformation rules (which triggers further research).

**Eliminating dead transitions**

Our naïve approach may easily generate redundant transitions with guards that can never be satisfied, although each one is investigated and tested at every step over and over again causing an unacceptable decline in performance. For instance, the finite automaton model in our running example (Fig. 2.3) has a single initial state $s_1$, therefore the transitions generated in Example 6.6 for $a_1-s_2$ and $a_1-s_3$ pairs are superfluous as the guards will constantly be false due to the static structure of the model.

To eliminate such an overhead, in Alg. 23, we eliminate transitions with guards that can never be satisfied by further pre-processing the TS representation of a graph transformation system.

The positive conditions $g_{pos} = \bigwedge_i p_i$ (generated in accordance with the LHS of the GT rule) are processed as follows (Lines 4–12).

- If a positive literal $p_i$ is constantly evaluated to false (i.e., it refers to some static parts which is not present in the initial model), then the corresponding transition $\tau$ is eliminated (Lines 5–6).
- If a literal $p_i$ is constantly evaluated to true (i.e., it refers to some static parts which is present in the initial model), then the guard of $\tau$ is truncated by removing $p_i$ (Lines 7–8).
- If the truth value of a literal $p_i$ may vary (as it is dynamic) then it is kept as it is in the guard of $\tau$ (Lines 9–10).

The negative conditions $g_{neg} = \bigwedge_i \neg\phi_i \equiv \bigwedge_i (\neg (\bigwedge_j p_{i,j})$ (generated in accordance with the NEG condition graph(s) of the GT rule) are processed follows (Lines 13–24).
6.3 From Graph Transformation Systems to Transition Systems

**Algorithm 23** Eliminating dead transitions in a TS

Notation:
\[ \tau \equiv g_{hs} \land g_{neg} : \rightarrow act \] a transition in the naive TS
\[ p_i : \text{a predicate in the LHS formula} \]
\[ \phi_i : \text{a (negated) conjunct of atomic predicates in NEG} \]
\[ [p_i]^{\text{init}} : \text{the truth value of } p_i \text{ in the initial state } \Xi^{\text{init}} \]

```python
func eliminate_dead_transitions =
  1: forall \( \tau \equiv (g_{hs} \land g_{neg} : \rightarrow act) \) generated by naive_transitions do
  2:      let \( g_{hs} = \bigwedge_i p_i \)
  3:      let \( g_{neg} = \bigwedge_i \neg \phi_i = \bigwedge_i (\neg (\bigwedge_j p_{i,j}) \]
  4:      forall \( p_i \in g_{hs} \) do
  5:          if \( p_i \) is static and \( [p_i]^{\text{init}} = \text{false} \) then
  6:              eliminate \( \tau \)
  7:          else if \( p_i \) is static and \( [p_i]^{\text{init}} = \text{true} \) then
  8:              remove \( p_i \) from \( g_{hs} \)
  9:          else if \( p_i \) is dynamic then
 10:              leave \( p_i \) in \( g_{hs} \) as it is
 11:      end if
 12:  end for
 13:  forall \( (\neg \phi_i) \in g_{neg} \) do
 14:      if \( \forall j : p_{i,j} \) is static and \( [p_{i,j}]^{\text{init}} = \text{true} \)
         (thus \( \neg \phi_i \) is \text{false}) then
 15:          eliminate \( \tau \)
 16:      end if
 17:      forall \( p_{i,j} \in \phi_i \) do
 18:          if \( p_{i,j} \) is static and \( [p_{i,j}]^{\text{init}} = \text{false} \) then
 19:              remove \( p_{i,j} \) from \( g_{neg} \)
 20:          else if \( p_{i,j} \) is dynamic then
 21:              leave \( p_{i,j} \) in \( \phi_i \) as it is
 22:      end if
 23:  end for
 24:  end for
 25: end for
```

- If for all \( j \) literals \( p_{i,j} \) are constantly evaluated to true, then a matching pattern of the NEG part of the rule is found which always prevents rule application thus we remove the entire transition \( \tau \) (Lines 14–15).
- If a literal \( p_{i,j} \) is constantly evaluated to false (Lines 18–19), then the current conjunct \( \phi_i \) of the negative condition can be removed from the guard. Since the application of the GT rule may still be prevented by another conjunct \( \phi_k \) thus we do not remove the entire negative condition in this step.
- If the truth value of a literal \( p_i \) may vary then it is kept as it is in the negative guard \( g_{neg} \) of \( \tau \) (Lines 20–21).

Note that since only dynamic elements can be modified therefore no further preprocessing is required for the actions of a guarded command. In other terms, a state variable array appearing in an action is guaranteed to be dynamic.

**Example 6.8** After this preprocessing step, we expect to have the following SAL specification for the transitions of our sample automaton model (in Fig. 2.3) formalized by graph transformation rules of Fig. 4.10.

```
% guarded commands for initR and reachableR
TRANSITION
  NOT reachable[sl][sl] ->
  reachable[sl][sl] = TRUE; [] % sl is init
```
reachable[a][s1] AND NOT reachable [a][s2] -->
reachable'[a][s2] = TRUE; \% s1 -> s2
reachable[a][s1] AND NOT reachable [a][s3] -->
reachable'[a][s3] = TRUE; \% s1 -> s3
reachable[a][s2] AND NOT reachable [a][s3] -->
reachable'[a][s3] = TRUE; \% s2 -> s3
END;

Starting from the naive encoding of initR of Example 6.6, on the one hand, we truncate the guard for the first potential match by eliminating literals automaton(a1), states(a1, s1), state(s1), etc. as they constantly evaluate to true. On the other hand, we eliminate the other two transitions since, init(a1, s2) and init(a1, s3) is constant false due to the static structure of the model.

Filtering properties

Since only dynamic elements appear in the optimized version of a TS generated from a graph transformation system, the (safety or deadlock) property to be verified should also be modified by removing the static parts. If we assume that a property does not contain (neither quantified nor unquantified) variables only specific locations (i.e., p(x) is not allowed if x is a variable only p(vi) where vi is a specific value), then we can proceed in a similar way as done when truncating/eliminating guards.

Example 6.9. Consider the following property of finite automata, which expresses that state s1 of automaton a1 is not reachable.

\[ \neg (\text{automaton}[a1] \text{ AND state}[s1] \text{ AND states}[a1][s1] \text{ AND reachable}[a1][s1]) \]

As the static parts are satisfied in the initial model, and they are not modified during the evolution of the model, the filtered property (containing only dynamic elements) is as follows.

\[ \neg (\text{reachable}[a1][s1]) \]

6.4 Proof of Operational Equivalence

In the current section, we show on the ASM level that there is bisimulation between the original graph transformation system and the generated transition system, moreover, there is a one-to-one mapping between the states of the corresponding ASMs. For the sake of simplicity, we split the proof into two (see Fig. 6.3).

- First we prove the equivalence of the ASM of a graph transformation system (ASM\textsuperscript{gr}) and the ASM of the naive encoding (ASM\textsuperscript{nve}) to show that our Cartesian product construction is appropriate (for unfolding graph transformation rules).
- Then we show that if we abstract from static parts of ASM\textsuperscript{nve} to obtain the optimized version of the target TS (ASM\textsuperscript{op}), this equivalence still holds after eliminating certain (dead) transitions.

6.4.1 Correctness and completeness of the naive encoding

For the first proof, we show that the ASM representation ASM\textsuperscript{gr} of a graph transformation system is equivalent (in a certain sense) with the ASM representation ASM\textsuperscript{nve} of a TS generated according to the naive encoding (Alg. 22).

For a notational shorthand, we write \( \mathfrak{A} = \mathfrak{B} \) if all locations are identical in states \( \mathfrak{A} \) and \( \mathfrak{B} \), i.e., for all function symbol \( p: \llbracket p(x) = v \rrbracket^\mathfrak{A} \iff \llbracket p(x) = v \rrbracket^\mathfrak{B} \).
6.4 Proof of Operational Equivalence

\[
\begin{align*}
&\mathcal{Q}^{opt} \xrightarrow{\mathcal{F}_2(S)} \mathcal{Q}^{opt} \\
&\begin{array}{c}
\mathcal{Q}^{nu} \\
\mathcal{Q}^{nu}
\end{array} \xrightarrow{\mathcal{F}_1(R)} \begin{array}{c}
\mathcal{Q}^{nu} \\
\mathcal{Q}^{nu}
\end{array} \\
&\begin{array}{c}
x_1 \\
x_1
\end{array} \xrightarrow{R} \begin{array}{c}
x_1 \\
x_1
\end{array}
\end{align*}
\]

Fig. 6.3. ASM abstraction/refinement scheme for proving the correctness of our model checking approach

**Proposition 6.10 (Equivalence of initial states).** The initial states \(\mathcal{Q}^{gr}_{init}\) of \(AS\mathcal{M}^{gr}\) and \(\mathcal{Q}^{nu}_{init}\) of \(AS\mathcal{M}^{nu}\) are equivalent with respect to \(\mathcal{F}_1\). Formally, \(\forall \mathcal{Q}^{gr}_{init} \exists \mathcal{Q}^{nu}_{init} : \mathcal{Q}^{nu}_{init} = \mathcal{F}_1(\mathcal{Q}^{gr}_{init})\), and \(\forall \mathcal{Q}^{nu}_{init} \exists \mathcal{Q}^{gr}_{init} : \mathcal{Q}^{gr}_{init} = \mathcal{F}_1(\mathcal{Q}^{nu}_{init})\).

**Proposition 6.11.** We can establish a bisimulation between the steps/runs and a bidirectional mapping between states of \(AS\mathcal{M}^{gr}\) and \(AS\mathcal{M}^{nu}\) (provided that nextIdFromResolve is implemented correctly). Formally,

1. **Completeness / Forward simulation.**
   \(\forall \mathcal{Q}^{gr} \forall \mathcal{Q}^{nu} \forall \mathcal{B}^{gr} : \mathcal{Q}^{nu} = \mathcal{F}_1(\mathcal{Q}^{gr}) \land \mathcal{B}^{gr} = nextR(\mathcal{Q}^{gr}) \rightarrow \exists \mathcal{B}^{nu} : \mathcal{B}^{nu} = next_1(\mathcal{Q}^{nu}) \land \mathcal{B}^{nu} = \mathcal{F}_1(\mathcal{B}^{gr})\).

2. **Correctness / Backward simulation.**
   \(\forall \mathcal{Q}^{gr} \forall \mathcal{Q}^{nu} \forall \mathcal{B}^{nu} : \mathcal{Q}^{nu} = \mathcal{F}_1(\mathcal{Q}^{gr}) \land \mathcal{B}^{nu} = next_1(\mathcal{Q}^{nu}) \rightarrow \exists \mathcal{B}^{gr} : \mathcal{B}^{gr} = nextR(\mathcal{Q}^{gr}) \land \mathcal{B}^{gr} = \mathcal{F}_1(\mathcal{B}^{nu})\).

**6.4.2 Correctness and completeness of the optimized encoding**

For the second part of the proof, we show that the ASM representation \(AS\mathcal{M}^{nu}\) of a TS is equivalent (in a certain sense) with the ASM representation \(AS\mathcal{M}^{opt}\) of a TS generated from \(AS\mathcal{M}^{nu}\) according to the optimization method by eliminating dead transitions in Alg. 23.

**Proposition 6.12 (Equivalence of initial states).** The initial states \(\mathcal{Q}^{nu}_{init}\) of \(AS\mathcal{M}^{nu}\) and \(\mathcal{Q}^{opt}_{init}\) of \(AS\mathcal{M}^{opt}\) are equivalent with respect to \(\mathcal{F}_2\). Formally, \(\forall \mathcal{Q}^{nu}_{init} \exists \mathcal{Q}^{opt}_{init} : \mathcal{Q}^{opt}_{init} = \mathcal{F}_2(\mathcal{Q}^{nu}_{init})\), and \(\forall \mathcal{Q}^{opt}_{init} \exists \mathcal{Q}^{nu}_{init} : \mathcal{Q}^{nu}_{init} = \mathcal{F}_2(\mathcal{Q}^{opt}_{init})\).

**Proposition 6.13.** We can establish a bisimulation between the steps/runs and a bidirectional mapping between states of \(AS\mathcal{M}^{nu}\) and \(AS\mathcal{M}^{opt}\). Formally,

1. **Completeness / Forward simulation.**
   \(\forall \mathcal{Q}^{nu} \forall \mathcal{Q}^{opt} \forall \mathcal{B}^{opt} : \mathcal{Q}^{opt} = \mathcal{F}_2(\mathcal{Q}^{nu}) \land \mathcal{B}^{opt} = nextR(\mathcal{Q}^{nu}) \rightarrow \exists \mathcal{B}^{nu} : \mathcal{B}^{nu} = next_2(\mathcal{Q}^{opt}) \land \mathcal{B}^{nu} = \mathcal{F}_2(\mathcal{B}^{opt})\).

2. **Correctness / Backward simulation.**
   \(\forall \mathcal{Q}^{nu} \forall \mathcal{Q}^{opt} \forall \mathcal{B}^{opt} : \mathcal{Q}^{opt} = \mathcal{F}_2(\mathcal{Q}^{nu}) \land \mathcal{B}^{opt} = next_2(\mathcal{Q}^{opt}) \rightarrow \exists \mathcal{B}^{nu} : \mathcal{B}^{nu} = nextR(\mathcal{Q}^{nu}) \land \mathcal{B}^{nu} = \mathcal{F}_2(\mathcal{B}^{opt})\).

As a consequence of all the propositions, we established the following main theorem for the correctness and completeness of our encoding.

**Theorem 6.14.** The initial states \(\mathcal{Q}^{gr}_{init}\) of \(AS\mathcal{M}^{gr}\) and \(\mathcal{Q}^{opt}_{init}\) of \(AS\mathcal{M}^{opt}\) are equivalent with respect to \(\mathcal{F}\) where \(\mathcal{F} = \mathcal{F}_2 \circ \mathcal{F}_1\). Formally, \(\forall \mathcal{Q}^{gr}_{init} \exists \mathcal{Q}^{opt}_{init} : \mathcal{Q}^{opt}_{init} = \mathcal{F}(\mathcal{Q}^{gr}_{init})\), and \(\forall \mathcal{Q}^{opt}_{init} \exists \mathcal{Q}^{gr}_{init} : \mathcal{Q}^{gr}_{init} = \mathcal{F}(\mathcal{Q}^{opt}_{init})\).

Moreover, we can establish a bisimulation between the steps/runs and a bidirectional mapping between states of \(AS\mathcal{M}^{gr}\) and \(AS\mathcal{M}^{opt}\) with the same \(\mathcal{F}\). Formally,
1. **Completeness / Forward simulation.**
   \[ \forall \mathcal{A}^g, \forall \mathcal{B}^g, \forall \mathcal{B}^{opt} : \mathcal{A}^{opt} = \mathcal{F}(\mathcal{A}^g) \land \mathcal{B}^g = \text{next}_R(\mathcal{A}^g) \rightarrow \exists \mathcal{B}^{opt} : \mathcal{B}^{opt} = \text{next}_R(\mathcal{A}^{opt}) \land \mathcal{B}^{opt} = \mathcal{F}(\mathcal{B}^g). \]

2. **Correctness / Backward simulation.**
   \[ \forall \mathcal{A}^g, \forall \mathcal{A}^{opt}, \forall \mathcal{B}^{opt} : \mathcal{A}^{opt} = \mathcal{F}(\mathcal{A}^g) \land \mathcal{B}^{opt} = \text{next}_R(\mathcal{A}^{opt}) \rightarrow \exists \mathcal{B}^g : \mathcal{B}^g = \text{next}_R(\mathcal{A}^g) \land \mathcal{B}^g = \mathcal{F}(\mathcal{B}^{opt}). \]

### 6.5 Dining Philosophers: A Case Study for Modeling and Verification

In the current section, our model checking framework is evaluated by a case study on the well-known problem of dining philosophers. Even though the problem itself is relatively simple from a modeling point of view, it frequently serves as a benchmark to assess the performance of verification tools.

For the case study, we will use only a single model checker tool, but the problem will be modeled and encoded into transition systems in different ways.

- First the dynamic behavior of dining philosophers will be captured by graph transformation rules, thus our transformation technique will be applied on the model-level.
- Then, the behavior of philosophers is formalized by UML statecharts and projected into transition systems according to a set of graph transformation rules providing formal semantics for statecharts on the meta-level.
- Finally, the UML statecharts description is projected directly (i.e., without the use of graph transformation) into transition system following the approach of [105].

By that case study, we try to assess the succinctness of the target transition systems (concerning the state space) in each case by increasing the number of philosophers, thus the different modeling formalisms will be judged from a verification point of view.

*The problem of dining philosophers*

In the dining philosophers’ problem, \( n \) philosophers are sitting around a table and thinking. From time to time, when they get hungry, they initiate an eating process by grabbing first a left fork and then a right fork. Unfortunately, there is a single fork between two philosophers (which is hence a shared resource) therefore they might need to wait for the forks to become available. When a philosopher manages to get both the left and the right fork, he (or she) starts eating immediately. As soon as the philosopher has finished eating he places back both forks and goes back to thinking.

From a verification point of view, our aim is to show that (started from a state where each philosopher is thinking and all forks are on the table) the system of dining philosophers will not reach a deadlock (deadlock freedom property), moreover, no forks are held at the same time by multiple philosophers (a safety criterion requiring mutual exclusion on forks).

#### 6.5.1 Case 1: Graph transformation on the model level

The class diagram in Fig. 6.4 captures the static structure of dining philosophers with two static classes (*Phi* and *Fork*) with two static associations *left* and *right* (identifying the left and right fork of a philosopher), a dynamic association *hold* (for expressing that a philosopher holds a certain fork), and the dynamic attribute *status* of philosophers, which may take the value from the enumeration *think*, *hungry, hasL* (has only left fork), *hasR* (has only right fork) and *eat*.

For the presentation in the current paper (but not for verification), we work with three philosophers in all three cases sitting around in a way depicted in the object diagram of Fig. 6.4.

The behavior of dining philosophers is captured in this case by the set of graph transformation rules shown in Fig. 6.5.
6.5 Dining Philosophers: A Case Study for Modeling and Verification

![Class diagram](image1)

![Object diagram](image2)

Fig. 6.4. Dining philosophers: class and object diagram

![Transformation rules](image3)

Fig. 6.5. Graph transformation rules for dining philosophers

- Rule `getHungryR` simply sets the `status` attribute of a thinking philosopher to hungry.
- A hungry philosopher may get his left fork by applying `getLeftForkR`, which requires that the left fork is not yet held as being the right fork of his/her right neighbor.
- A philosopher already having the left fork in hand may apply for the right fork by executing rule `getRightForkR` (which is conceptually similar to `getLeftForkR`) and starts eating if succeeded.
- An eating philosopher will release his or her left fork at some point by applying rule `finishEatingR` getting into the status `hasR` in the meantime.
- Finally, if a philosopher only has the right fork in hand, then the fork can be safely released, and the philosopher goes back to thinking (see rule `releaseRightForkR`).

The corresponding transition system derived according to our encoding of Sec. 6.3 is listed in Appendix C.1.

Naturally, the verification process automatically detects that the system may get into a deadlock, if each philosopher only manages to grab his or her left fork and thus waits for the right fork forever. However, our safety property requiring that no fork is held by two philosopher at any time (formally, \( \forall f: \text{Fork}, ph_1, ph_2: \text{Phil}: G \ ( ph_1 \neq ph_2 \rightarrow \neg (\text{hold}[ph_1][f] \land \text{hold}[ph_2][f])) \)) is verified.

The two traditional ways to avoid the deadlock problem for dining philosophers are depicted in Fig. 6.6.
• An additional graph transformation rule \((\text{releaseLeftForkR})\) can be introduced to move the system from a possibly deadlock situation by prescribing that if a philosopher already having a left fork cannot get the right fork as well then the left fork is put back onto the table and the philosopher goes back to hungry status.
• Alternatively, rules \(\text{getLeftForkR}\) and \(\text{getRightForkR}\) can be merged into a single rule \(\text{getBothForksR}\), when both left and right forks are grabbed at once thus preventing the philosopher to wait for a single fork.

The verification of the transition systems generated from these two versions proved that no deadlock situation is possible (while the safety criterion is still satisfied); however, there were be crucial differences in performance.

In a traditional graph transformation system, the first solution would typically supercede the other in performance as the application of a more complex rule takes much more time than formalizing the problem with a set of rules of smaller complexity (by complexity we mean the number of nodes in the \(Lhs\) and \(Neg\) graphs since the performance of pattern matching is the critical phase in rule application).

However, after all the preprocessing (collecting potential matchings) performed at compile time, it will turn out that having a small number of relatively complex rules yield a better performance for verification than a larger number of rules with relatively low complexity (see Table 6.2 for a comparison later in Sec. 6.5.3).

6.5.2 Case 2: Graph transformation on the meta-level

In our second case study, graph transformation is applied on the meta-level: the dining philosophers’ problem is captured by means of UML Statecharts (see 6.7) on the model-level but the formal semantics of statecharts is formalized by graph transformations rules (as depicted in Fig. 6.9 and 6.10).

We applied the following restrictions on the traditional UML statecharts to achieve a solution that is comparable to the one obtained from model-level graph transformation rules.

• All the actions are considered to be send actions, and each transition may only contain a single send action as the effect.
- Send actions are addressed by role names appearing in the class diagram (as a statechart is a class-level specification mechanism).
- Guards may only contain a single $\text{ISIN}(\text{rolestate})$ statement querying whether another statemachine accessible via the specific role stays currently in a certain state.

*Behavior of philosophers expressed by UML Statecharts*

Therefore, a thinking philosopher may get into a *hungry* state at any time. After that, he checks whether his left fork is in the *free* state, and if so, sends an acquire (*acq*) message to the *left* fork and moves himself to *hasLeft* state. Next, the same procedure is done for acquiring the right fork getting the philosopher into the *eating* state. Finally, after eating, the forks are released one by one, by sending a *rel* message to the *left* fork and the right fork, respectively.

The statemachine of a fork simply contains two states (*free* and *held*) stating whether the fork is held by a philosopher or it is situated on the table. Transitions between them are triggered by the *acq* and *rel* messages, respectively.

*Semantics of UML Statecharts by graph transformation*

To provide formal semantics for UML Statecharts by graph transformation systems, we have built on [168] where an extended hierarchical automaton formalism was used as the underlying structural representation for statecharts. However, for the current case study, the state hierarchy was flattened by collecting all state configurations and transitions that can be fired simultaneously at a preprocessor phase (as done in [105]), thus obtaining a flat finite automaton formalism (see the metamodel in Fig. 6.8) communicating with message passing to each other as a simplification.

![UML Statechart Diagram for the Dining Philosophers Problem](image-url)

**Fig. 6.7.** Dining philosophers defined by UML Statecharts

![Flat Finite Automaton for UML Statecharts](image-url)

**Fig. 6.8.** A metamodel of flattened UML statecharts
• The static parts of the metamodel are the following:
  - The automaton \( A \) consists of state configurations (\( Config \)) and steps (\( Step \), which are the collections of transitions that can be fired at a time).
  - Each step has a source \( src \) and a target \( trg \) configuration, an optional \( inState \) link to the configuration of another automaton (ISIN statement), a trigger \( Event \), and an \( Action \) as effect.
  - An \( Action \) contains a link to an \( Event \) to be sent and a \( receiver \) automaton.

• Dynamic parts of the metamodel are constituted as follows:
  - Each automaton may have an \( isAct \) link pointing to a configuration indicating that the automaton is currently in the certain configuration,
  - An automaton also has an \( inQueue \) link indicating whether a specific event is in the event queue of the automaton (event queues are modeled as sets, thus only one instance of the event may be stored in the queue).
  - The \( fire \) link leading to a \( Step \) states that the automaton is currently firing all the transitions in \( Step \).
  - Finally, \( pc \) is a simple program counter indicating which phase of the statechart semantics is being executed for the moment by the virtual state machine.

The graph transformation rules in Fig. 6.9 handle state (configuration) changes of a state machine identifying four different cases split on the basis of existence of \( inState \) links and \( trigger \) events.

• In each case, the \( isAct \) link of the automaton \( A_i \) is rewritten from configuration \( S_1 \) to configuration \( S_2 \) supposing that the two configurations are connected by a step \( T_1 \). Meanwhile, a \( fire \) link is added leading to \( T_1 \) (since the step \( T_1 \) is being fired currently), and the \( pc \) attribute of the automaton is updated to \( addQR \) to indicated that before processing the next event, the automaton must send the associated actions (to provide a synchronized behavior of the machine).

• In each four cases, there are additional conditions that are required for executing the current step of the state machine (automaton).
  - Rule \( fireNoEvtWithinStr \) handles the case when that no trigger events are associated to the step (see the negative condition), however, the configuration identified by the \( inState \) link must be active in the corresponding automaton.
  - Rule \( fireWithEvtWithinStr \) requires the trigger event of the step to be in the event queue of the automaton (the \( inQueue \) link), moreover, the configuration identified by the \( inState \) link must be active in the corresponding automaton. As an additional result, the event is removed from the event queue.
  - Rule \( fireNoEvtNonStr \) forbids the existence of both a trigger event and an \( inState \) link in the context of the step.
  - Rule \( fireWithEvtNonStr \), finally, prescribes the non-existence of an \( inState \) link but requires that the trigger event of the step to be in the event queue of the automaton (\( inQueue \) link). As an additional result this time as well, the event is removed from the event queue.

Message sending is modeled by graph transformation rules of Fig. 6.10.

• In case of rule \( addQueueWithAct \) there is a send action associated to the step, thus the event of the action should be placed in the event queue of the \( receiver \) automaton as a result of rule application. Note that the rule is only applicable if a certain step has already been selected to firing (see the \( fire \) link and the \( addQR \) value of the program counter).

• In case of rule \( addQueueNoAct \) there are no actions related to the firing step thus the rule simply concludes that the firing of the transition has terminated (thus, in both cases, the \( fire \) link is removed and the program counter is updated to the \( fire \)).
Fig. 6.9. Firing a transition (set) in a statechart automata

In fact, this statechart semantics is probably oversimplified (not fully realistic) when compared to the original UML semantics, but the formalism had to be kept simple in order to provide some comparison with the other case studies from a verification point of view. For more precise ways of formalizing UML statecharts by graph transformation systems the reader is referred to [101, 168].
As graph transformation techniques were applied on the meta-level, the derived transition system (enlisted in Appendix C.2) is totally different in this case, since the state variable arrays are defined by the (meta)classes of the metamodel (in Fig. 6.8), and not by the classes of the model (Fig. 6.4).

Even though for meta-level encodings the verification managed to terminate only for few number of philosophers (as shown in Table 6.2), the verification process automatically detected a non-trivial error in the our graph transformation semantics of UML statecharts which leads our system into a state where our safety criterion is violated.

The problem originates from the fact that testing whether the fork is not held by anyone, and actually acquiring the fork is not an atomic operation. Moreover, the system may evolve between the sending of an acq message and the processing of the same message by the receiver since UML assumes a distributed environment for the execution of statemachines. Therefore, two philosophers may find the same fork to be free at one step and sending only the acquire message in the other. As a result, both philosophers move to a state when they think that they hold the specific fork, which is a contradiction with our safety requirement.

We can now conclude that (i) either the graph transformation rules presented as semantics for UML statecharts (in Fig. 6.9 and 6.10) are not appropriate, as the well-known system of dining philosophers behaves differently than what was expected (a validation problem), or (ii) if we accept that the transformation rules precisely capture the semantics of UML statecharts, then the UML specification of dining philosophers is not correct (a verification problem).

Model-level encoding of UML Statecharts

In order to provide comparison with an existing approach for verifying UML statecharts, we adapted the SPIN encoding of UML statemachines described in [105] for other target model checkers. The main idea in the approach is to maintain only a single state variable for the state configuration information of each object and a state variable for its event queue in addition. Therefore, when a step is fired, the state changes and the message sendings are performed in a single and atomic operation (however, the synchronization of sending and receiving acquire messages had to be solved on the UML model level). The model-level (direct) encoding of UML statecharts representation of dining philosophers is listed in Appendix C.3.
6.5 Dining Philosophers: A Case Study for Modeling and Verification

6.5.3 Assessment of verification results

The three different kinds of specifications were executed by a model checker with different number of philosophers. Since the SAL model checker is not publicly available yet, for the concrete verification runs we used the Murϕ system to increase the comparability of results.

For our test experiments (summarized in Table 6.2), for each verification run at most 100 Megabytes of system memory was allocated to store the state space, and Murϕ was running on a 550 MHz Pentium III machine.

<table>
<thead>
<tr>
<th>Test case</th>
<th>N</th>
<th>States</th>
<th>Fired tr.</th>
<th>Time</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT (err)</td>
<td>5</td>
<td>235</td>
<td>405</td>
<td>0.12</td>
<td>dead</td>
</tr>
<tr>
<td>GT (ok1)</td>
<td>3</td>
<td>75</td>
<td>201</td>
<td>0.15</td>
<td>ok</td>
</tr>
<tr>
<td>GT (ok1)</td>
<td>5</td>
<td>1365</td>
<td>6095</td>
<td>0.18</td>
<td>ok</td>
</tr>
<tr>
<td>GT (ok1)</td>
<td>9</td>
<td>430403</td>
<td>3535515</td>
<td>12.29</td>
<td>ok</td>
</tr>
<tr>
<td>GT (ok2)</td>
<td>3</td>
<td>20</td>
<td>48</td>
<td>0.10</td>
<td>ok</td>
</tr>
<tr>
<td>GT (ok2)</td>
<td>5</td>
<td>152</td>
<td>620</td>
<td>0.10</td>
<td>ok</td>
</tr>
<tr>
<td>GT (ok2)</td>
<td>12</td>
<td>1722928</td>
<td>16953600</td>
<td>103.12</td>
<td>ok</td>
</tr>
<tr>
<td>GT/SC (err)</td>
<td>3</td>
<td>931</td>
<td>2387</td>
<td>0.15</td>
<td>unsafe</td>
</tr>
<tr>
<td>GT/SC (err)</td>
<td>5</td>
<td>6287</td>
<td>20519</td>
<td>0.49</td>
<td>unsafe</td>
</tr>
<tr>
<td>GT/SC (ok)</td>
<td>2</td>
<td>68084</td>
<td>236902</td>
<td>6.59</td>
<td>ok</td>
</tr>
<tr>
<td>SC (err)</td>
<td>3</td>
<td>743</td>
<td>1915</td>
<td>0.98</td>
<td>dead</td>
</tr>
<tr>
<td>SC (err)</td>
<td>5</td>
<td>128439</td>
<td>577570</td>
<td>12.52</td>
<td>dead</td>
</tr>
<tr>
<td>SC (ok)</td>
<td>3</td>
<td>7057</td>
<td>29289</td>
<td>1.43</td>
<td>ok</td>
</tr>
<tr>
<td>SC (ok)</td>
<td>4</td>
<td>138001</td>
<td>764936</td>
<td>16.24</td>
<td>dead</td>
</tr>
</tbody>
</table>

There were seven different specifications tested with an increasing number of philosophers. For each specification, the last line of the test cases contain the maximum number of philosophers for which the verification terminated within the given resources.

The columns of the table contain, respectively, (1) the identifier of the specification, (2) the number of philosophers, (3) the number of states traversed, (4) the number of transitions during verification, (5) an average execution time (in seconds), and (6) the result of the verification (where ‘unsafe’ means that the safety criterion was violated, ‘dead’ refers to the fact that a deadlock was detected, while ‘ok’ means that no errors were found).

The different specifications are encoded as follows:

- **GT (err)**: the model-level encoding of the dining philosophers problem using the original set of graph transformation rules of Fig. 6.5;
- **GT (ok1)**: the model-level encoding of the dining philosophers problem using a corrected set of graph transformation rules with rule releaseLeftForkR of Fig. 6.6;
- **GT (ok2)**: the model-level encoding of the dining philosophers problem using a corrected set of rules including rule getBothForksR of Fig. 6.6;
- **GT/SC (err)**: the meta-level encoding of the dining philosophers problem captured by the UML statecharts of Fig. 6.7 (with statechart semantics defined by graph transformation rules of Fig. 6.9 and 6.10);
- **GT/SC (ok)**: the same approach as before but corrected by certain changes on the statechart (not discussed in the paper in details);
- **SC (err)**: the direct (model-level) encoding of UML statecharts to Murϕ following the guidelines of [105]. This test set is identical to the case captured by statechart of Fig. 6.7;
- **SC (ok)**: the same as before but an additional transition was introduced leading back to hungry state from hasLeft to get out from a possibly deadlock situation (following the principles of rule releaseLeftForkR on the statechart model).
From these verification results, the following conclusions can be drawn.

**Remark 6.15.** Transition systems derived from graph transformation systems used as *model-level* specifications by Alg. 22 and 23 show very good performance in verification (comparable to manual encoding of the problem as a transition system).

We can also derive that using our encoding on the meta-level has certain practical limitations on the size of the model to be verified.

**Remark 6.16.** The verification of transition systems derived from graph transformation systems used as a *meta-level* specification technique would typically work for small models of the modeling language in question. However, meaningful specification flaws (either in the model or in the formal semantics) can frequently be detected even on such relatively small models.

Typically, when the semantics of a new modeling language is created, it is first tested on *very small models*, thus a large percentage of weaknesses could possibly be detected by our technique.

Moreover, our recent investigations show that SPIN provides better facilities to handle the interleaving of transitions (based on partial orderedness and the explicit use of queues for communication between processes) than Murφ did in our case study.

**Remark 6.17.** Graph transformation systems with a small number of complex rules typically behaves better for verification than a graph transformation system with a larger number of relatively simple rules formalizing the same problem.

This statement is a result of the our compile time preprocessing step that collects all potential matches thus the target transition system used for verification does not have to execute complex queries on the model. Therefore, (unlike the case of running a simulation of a system in a traditional graph transformation tool) a smaller set of complex rules would cause fewer interleavings of transitions resulting in better run-time performance.

**Additional case studies**

These conclusions drawn from the dining philosophers problem was also supported by additional simple verification case studies that have been carried out within the SAL environment by encoding UML statecharts into SAL specifications. In fact, the need for the optimizations described in Sec. 6.3 were triggered by unsuccessful preliminary verification attempts. The automatic transformation into SAL specifications (for this specific modeling language, namely, UML) was carried out within the VIATRA environment [191].

In another case study of our approach, we captured the operational semantics of Petri nets by graph transformation systems and translated them into the specification language of the Murφ model checker. In case of bounded Petri nets (where the number of tokens in Petri nets has an *a priori* upper bound), our approach was directly applicable. We also exploited the use of predicate abstraction by abstracting away from the concrete number of tokens, which resulted in a semi-decision procedure for proving liveness and safety properties of Petri nets.

Recently, our technique was applied to carry out formal analysis of architectural styles [16,17] in an early phase of design. For these investigations, we proved reachability properties (i.e., to decide whether certain target states are reachable or not from a given initial state) of a heavily dynamic GTS (with several dynamic classes and associations).

**6.6 Conclusions and Future Work**

I presented an approach for the automated verification of any specific instance model of an arbitrary modeling language (with static structure defined by metamodeling and operational semantics defined by graph transformation systems) using existing model checker tools.
• **Provenly correct transition system representation.** For any graph transformation system I describe an encoding (Sec. 6.3 based on [16,17,146,147,170–172]) which derives a behaviorally equivalent transition system, thus allowing the formal and automated verification of graph-based models against consistency properties (such as reachability and safety) using model checker tools.

• **Optimized model description.** In order to avoid state space explosion frequently experienced in the model checker tools due to the large size of the model, I introduced several optimizations (Sec. 6.3.3 following [147,171,172]) in the transition system encoding of a graph transformation system. These optimizations exploit that fact that graph transformation rules only modify (by definition) dynamic model elements thus all the static parts of a model can be eliminated after a compile-time preprocessing phase.

• **Performance analysis of verification.** I demonstrated the practical feasibility of the approach by assessing the efficiency of model checking the target transition system on benchmark verification problems (Sec. 6.5 based on [172]). As a conclusion, I derived the "few complex is better than many simple" principle as a verification rule of thumb which may help the transformation engineers to design graph transformation systems which are efficient from a verification aspect.

**Conceptual relevance**

The presented approach is simultaneously applicable for encoding (i) well-formed models of arbitrary modeling languages with semantics defined by metamodeling techniques (abstract syntax) and graph transformation rules (operational semantics) and (ii) graph transformation systems (in the original model-level sense) for describing the dynamic behavior of user models into transition systems.

As a result, we are able to verify semantic properties (like safety, deadlock freedom, etc.) of any specific well-formed model instance of the language or the user model itself by off-the-shelf model checking tools provided that their specification language is based on transition systems. This way, traditional correctness results from the theory of graph transformation (like confluence [45]), which provide solutions for proving very general properties of a specific problem can be complemented by our technique to reason about problem-specific properties by existing model checker tools.

As a verification rule of thumb for graph transformation systems, we established the "few complex is better than many simple" principle when concerning the complexity (of the left-hand side and negative condition graphs) of rules. This is a relatively unexpected result, since simulation tools for graph transformation are less efficient when handling rules of large complexity. The reason for this new principle lies in the fact that the static part of the pattern matching process is carried out at compile-time in a preprocessing step which is naturally not measured when assessing the performance of the model checker tool.

**Feasibility of the approach**

The feasibility of our approach was demonstrated on a well-known verification benchmark, i.e., by verifying deadlock freedom and a safety property for the dining philosophers' problem captured by graph transformation both on the model-level and the meta-level. However, note that our technique also allows to investigate liveness or reachability properties as demonstrated by additional case studies.

It is worth emphasizing that my approach is the first language independent solution for the model checking problem for graph transformation systems with performance assessment of the model checking process on various benchmarks.

**Practical relevance**

Since the feasibility of my approach was demonstrated for small and medium size models, these conceptual advances have become practical advances as well.

Furthermore, the advantage is more striking from the tool building point of view. Traditionally, bridging a new modeling language (such as new UML dialects turned up in UML 2.0) and a model
checker required a precise knowledge of both the dynamic semantics of the modeling language and the technicalities of the model checker tool. The proposed technique allows the verification engineers to concentrate only to the semantics of the modeling language, since the model checking description can be derived as soon as the semantics of the language is captured by a set of graph transformation rules.

**Tool support**

After having carried out several benchmark experiments in different domains (with partially automated translations), we are currently building a tool [147] that is capable of automatically translating models of arbitrary visual modeling languages defined by metamodeling and graph transformation (and thus model-level specifications based on graph grammars as well) into the corresponding Promela specifications (which is the input language of the SPIN model checker).

Again, initial experiments demonstrate that the compile time preprocessing of graph transformation rules driven by the metamodel and the instance model is more efficient then the subsequent model checking phase.

**Limitations**

The main conceptual limitation of the approach is that the instance model has to be a priori bounded, i.e., we carry out bounded model checking [39] on graph transformation systems. On the other hand, the performance capabilities of model checkers are imposing the main limitations imposed on my technique.

**Future work**

However, further research is required to overcome the state explosion problem we experienced in meta-level encodings of models (and, naturally, in model-level encodings of complex IT systems). Our case study has clearly revealed that the bottleneck of the verification process is not merely the number of transitions generated by our algorithm, but rather the interleaving of different transitions (i.e., larger steps in the evolution of the model yield a smaller number of intermediate states).

The main problem originates from the fact that traditional confluence results of the graph transformation theory cannot be exploited during model checking. Since in many cases, the behavior of the system is required to be investigated only on a single path instead of all possible interleavings of rule applications due to an appropriate theorem or analysis technique (like, e.g., critical pair analysis [36, 85]). Unfortunately, existing model checkers typically do not provide any facilities for the user to control the model checking process in such a way. For instance, in meta-level encodings, certain intermediate states of the graph transformation system (like executing a step and sending the action in case of our statecharts semantics) should be transparent for the model checker (without considering interleavings of transitions for them).

As a consequence of this reduced controllability, control structures appearing in many existing graph transformation tools (like, for instance, in case of PROGRES [151] or VIATRA [191]) can only be encoded as data (thus in the state variables as was done in our Petri net case study), which rather increases the number of interleavings instead of drastically decreasing them.

A different line of research aims at adapting our encoding to verification tools using labeled transition systems (LTS) as the underlying mathematical formalism (where information is related to transitions in contrast to Kripke structures where it is stored in states) in order to investigate process algebras (like CSP [86], or CCS [113]). Here the practical goal is to consistency analysis of behavioral UML diagrams extended with graph transformation rules in the style of [63].

Further challenges involve the formal analysis of graph transformation systems with time [76] based upon model checkers like Kronos [51], which typically use timed automata [6] as their specification language. As a potential result, we aim at investigating real-time systems specified in a high-level visual notation.
Towards Automated Formal Analysis of Model Transformations

I propose automated means to formally reason about the (i) syntactic (static) correctness and completeness of model transformations by using planner algorithms, and (ii) preservation of semantic (behavioral) properties by applying model checking techniques.

7.1 Formal Analysis: Why and What?

Unfortunately, the design of model transformations between modeling languages in a UML environment can be rather complicated thus error prone. Therefore, prior to reasoning about the UML model of the target application, we have to prove that the model transformation itself is free of conceptual flows.

However, there is not a single notion of correctness for model transformations: we can only identify typical questions to ask.

- Initially, one can prove that the result of the transformation is a well-formed instance of the target language. However, this is a syntactic issue, and **syntactic well-formedness** rarely guarantees the overall semantic correctness of a transformation.
- As UML does not have a formal semantics, many transformations from UML as the source language can only be **validated** against our informal expectations.
- To avoid such problems, one can transform UML models into a semantic domain, and after that, back-end transformations (from this semantic domain to the target mathematical analysis formalism) can be formally **verified**.

7.1.1 Correctness criteria of transformations

The most elementary requirements of a model transformation are syntactic.

- The minimal requirement is to assure **syntactic correctness**, i.e., to guarantee that the generated model is a syntactically well-formed instance of the target language.
- An additional requirement (called **syntactic completeness**) is to completely cover the source language by transformation rules, i.e., to prove that there exists a corresponding element in the target model for each construct in the source language.

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

- **Correctness (Dynamic consistency):** As model transformations may also define a **projection** from the source language to the target language, semantic equivalence between models cannot always be proved. Instead we define correctness properties, which are typically transformation
specific. For instance, in our EHA2PN benchmark transformation (Sec. 8.3), one may prescribe the natural criterion for correctness that each configuration that is reachable from the initial configuration in an EHA model should have a reachable counterpart (i.e., a marking) in the Petri net equivalent.

- **Termination:** We must also guarantee that a model transformation will terminate. A non-terminating transformation can be caused by a cycle in the control flow graph with unsatisfiable termination condition or an erroneous use of the loop mode for a rule.

- **Uniqueness (Confluence):** As non-determinism is used in the specification of model transition systems (selecting an appropriate matching for rule applications) we must also guarantee that the transformation yields a unique result.

Note that the formalization of these criteria for a specific model transformation is not at all straightforward. In many cases, we can reduce the question to a reachability problem or a safety property, but even in this case finding the appropriate temporal logic formulae is non-trivial.

**Related work**

The questions of static (or syntactic) completeness and correctness (consistency) of model transformations have recently become a hot topic in a UML environment (see e.g. [80]).

The questions of termination and confluence (uniqueness) problem have been thoroughly investigated in general by the theory of graph transformation systems (see, for instance, [142] for an overview). Unfortunately, many of these theoretical results are negative, e.g., the termination of a graph transformation system is undecidable in general [137].

Therefore, the main line of research has been conducted to define sufficient conditions for termination and confluence. For instance, critical pair analysis [36,85] provides a static analysis technique to detect conflicting rules that might violate confluence. A recent approach [102] identifies certain sufficient conditions for the uniqueness and termination of model transformations.

Dynamic consistency properties of UML models were mechanically analyzed in [62]. In theory, CSP may provide a semantic domain for other modeling languages as well, however, defining semantics of modeling languages in this way is a complicated task and requires to design complex model transformations. However, no complex solutions have been proposed to formally analyze that arbitrary model transformations between any pair of modeling languages preserve certain dynamic consistency properties.

**Objectives**

In the current chapter, we first investigate the questions of syntactic correctness and completeness (in Sec. 7.2), and then provide model-level but language independent means to verify semantic correctness (in other terms, dynamic consistency) properties of arbitrary transformations (Sec. 7.3). On the other hand, concerning termination and uniqueness, we rely on existing theoretical results and suppose that a non-conflicting set of model transformation rules guarantees the uniqueness of a model transformation (as it happens in many practical cases).

As an overview, we present the information flow of the chapter in Fig. 7.1.

### 7.2 Syntactic Correctness and Completeness of Transformations

In this section, the concepts of syntactic correctness and completeness for model transformation systems (called syntactic well-formedness in the sequel) will be defined on two levels.

- **Model dependent approach:** In this first case, the well-formedness of individual transformation instances (i.e. the transformation of a specific source model) are checked.
7.2 Syntactic Correctness and Completeness of Transformations

7.2.1 Model correctness
7.2.2 Transformation correctness
7.2.3 Examples on correctness and completeness
7.2.4 Proving correctness by planner algorithms

7.3 Automated Formal Verification of Model Transformations
7.3.1 Conceptual overview
7.3.2 Metamodel vs. model level verification

2.2 Specifying the Abstract Syntax of Modeling Languages
4.4.2 Defining modeling languages: dynamic behavior
4.4.4 Model transformation systems
6.3 From Graph Transformation Systems to Transition Systems

Fig. 7.1. Information flow in Chapter 7

- **Metamodel/grammar dependent approach:** In this case, we are aiming to prove that the transformation is correct for any instance of the source metamodel (i.e. any sentence of the source grammar).

Before being able to discuss the correctness of model transformations, one has to decide what a correct model and a correct transformation is.

- Correctness is defined by means of visual languages and graph grammars in analogy with traditional computational linguistics (thus on a **syntactic level**).
  - **Model dependent (simple) correctness** is defined by means of parsing the visual sentences generated by a model transformation by using the graph grammar of the target language.
  - **Metamodel dependent (total) correctness** is stronger than the previous as it aims to prove that each model generated by a model transformation system is a sentence of the target language. As the structure of model transformation rules mainly resembles to the structure of the source model, this problem is not at all trivial.

For the rest of this section, we suppose that a graph grammar exists for each metamodel which controls the construction of well-formed visual sentences. As the process of editing such models is not considered, we may also suppose that no rules of these graph grammars prescribe the deletion of some elements. Naturally, the structure of model transformation rules is not restricted in this sense.

Below, we first formalize syntactic correctness and completeness and then present a technique based on planner algorithms for constructively proving the syntactic correctness and completeness of model transformations. The technicalities will be demonstrated on a running example, which is a simplified fragment of a complex UML model transformation (to be introduced in Sec. 7.2.3).

### 7.2.1 Model correctness

Thus, in the current section, a model is considered to be correct whenever it can be derived from the start graph (regarded as an axiom) by graph transformation rules (regarded as deduction rules) defining the visual grammatical structure.

**Definition 7.1 (Graph grammar).** Let $\mathcal{G} = (\Sigma, \mathcal{R}_\Sigma)$ be a production system with the axiom $\Sigma$ (start graph) and deduction rules $\mathcal{R}_\Sigma$ (graph transformation rules). This system is called a **graph grammar** (considering that all graph nodes are terminal nodes).

**Definition 7.2 (Derivable).** Let $\mathcal{G} = (\Sigma, \mathcal{R}_\Sigma)$ be a graph grammar. We call a graph $\mathcal{M}$ (called model or sentence later) **derivable** from $\mathcal{G}$ (denoted as $\mathcal{G} \vdash \mathcal{M}$) iff $\mathcal{M}$ can be obtained from the start graph $\Sigma$ by a finite sequence of graph transformation steps using deduction rules $\mathcal{R}_\Sigma$. 
Definition 7.3 (Visual language). Let $\mathcal{G}$ be a graph grammar. The visual language (of the graph grammar), denoted as $\mathcal{L}_G$, contains all the graphs that are derivable from $\mathcal{G}$.

$$\mathcal{L}_G = \{ M \mid \mathcal{G} \vdash M \},$$

where $M$ is graph called (visual) sentence.

7.2.2 Transformation correctness

After discussing the correctness of models, correctness of model transformation will be introduced, built upon well-formed source models as axioms. Note that we revisit the original definitions (with slight modifications to improve presentation) from Sec. 4.4.4.

Definition 7.4 (Model transformation (revisited)). Let $\mathcal{T} = (M_A, \mathcal{R}_T)$ be a graph grammar with a start graph $M_A$, called source model (a sentence of the source visual language), and model transformation rules $\mathcal{R}_T$. $\mathcal{T}$ is denoted as model transformation.

Definition 7.5 (Model transformation system (revisited)). A model transformation system is a tuple $\mathcal{MTS} = (A, \mathcal{R}_T, B)$, where $A$ and $B$ are graph grammars defining the source and target language, respectively, and $\mathcal{R}_T$ is a set of model transformation rules.

Corollary 7.6. Let $\mathcal{MTS} = (A, \mathcal{R}_T, B)$ be a model transformation system. $\forall M_A : A \vdash M_A$, $\mathcal{T} = (M_A, \mathcal{R}_T)$ is a model transformation.

Several models may be derived by a model transformation system from different source models. However, such a system is of little importance if these derived models are incorrect sentences of the target graph grammar. To express the difference, models that are derived by a model transformation system will be denoted as model candidates. This concept of correctness is illustrated in Fig. 7.2.

![Concepts of correctness](image)

**Fig. 7.2. Concepts of correctness**

Definition 7.7 (Derivable as target). Let $\mathcal{T} = (M_A, \mathcal{R}_T)$ be a model transformation. A graph $M_B$ (called model candidate) is derivable as target from $\mathcal{T}$ (denoted as $\vdash \mathcal{T} M_B$ or $\{ M_A, \mathcal{R}_T \} \vdash \mathcal{T} M_B$) iff

$$\exists M_C : (T \vdash M_C) \wedge (M_A \cup M_B = M_C \setminus \Omega),$$

where $\Omega$ is the set of all reference nodes and edges in $M_C$.

Model transformation rules build a common supergraph $M_C$ containing the original source model $M_A$ and the novel target model candidate $M_B$, which are connected by reference nodes and edges $\Omega$. The target model candidate can be obtained from this supergraph if the original source model and the reference objects are removed.

In the following, the most important notions of model transformation systems (namely, correctness and completeness) are defined.

- Informally, a *model transformation is correct*, if the derived target candidate is a sentence of the target language (model dependent).
- A *model transformation system is correct*, whenever correctness holds for each source model (meta-model dependent).
Definition 7.8 (Correctness: model transformation). Let \( \mathcal{T} = (\mathcal{M}_a, \mathcal{R}_\tau) \) be a model transformation and \( \mathcal{B} \) be a graph grammar defining the target language. \( \mathcal{T} \) is correct (with respect to \( \mathcal{B} \)) iff
\[
\mathcal{T} \vdash \mathcal{M}_B \Rightarrow \mathcal{B} \vdash \mathcal{M}_B.
\]
Such a correctness will also be denoted as simple correctness.

Definition 7.9 (Correctness: model transformation system). Let \( \mathcal{MTS} = (\mathcal{A}, \mathcal{R}_\tau, \mathcal{B}) \) be a model transformation system. \( \mathcal{MTS} \) is correct iff
\[
\forall \mathcal{M}_a : \mathcal{A} \vdash \mathcal{M}_a \land \{ \mathcal{M}_a, \mathcal{R}_\tau \} \vdash \mathcal{M}_B \Rightarrow \mathcal{B} \vdash \mathcal{M}_B.
\]
Such a system is also called total correct.

Completeness is also defined on two levels: finding an appropriate source model for a given target sentence and for each target model.

Definition 7.10 (Completeness: model transformation). Let \( \mathcal{B} \) be a graph grammar, and \( \mathcal{M}_B \) be a model of this grammar. A model transformation \( \mathcal{T} \) is complete (with respect to \( \mathcal{M}_B \)) iff
\[
\mathcal{B} \vdash \mathcal{M}_B \Rightarrow \mathcal{T} \vdash \mathcal{M}_B.
\]
This process (finding one or more source models for a given target sentence) is also called back-annotation or simple completeness.

Definition 7.11 (Completeness: model transformation system). Let \( \mathcal{MTS} = (\mathcal{A}, \mathcal{R}_\tau, \mathcal{B}) \) be a model transformation system. \( \mathcal{MTS} \) is complete iff
\[
\forall \mathcal{M}_B \exists \mathcal{M}_a : \mathcal{B} \vdash \mathcal{M}_B \Rightarrow \{ \mathcal{M}_a, \mathcal{R}_\tau \} \vdash \mathcal{M}_B \land \mathcal{A} \vdash \mathcal{M}_a.
\]

7.2.3 Examples on correctness and completeness

In order to demonstrate the technicalities of proving syntactic correctness and completeness of model transformations, we selected a small fragment of a complex transformation as the basis of our running example. The complete transformation (discussed in detail conceptually in [26] and on the rule level in [177]) generates stochastic Petri nets from static UML models enriched with special dependability attributes (e.g., failure rate of components). Each static relation between high-level objects is regarded as a potential error propagation path. This Petri net based analysis aims at the identification of dependability bottlenecks in an early phase of design.

The entire transformation is divided into two major steps.

1. At first, an Intermediate Model (in the form of a simple hypergraph) is derived in order to extract important dependability attributes from UML models.
2. Afterwards, the Petri net model can be transformed straight from this intermediate hypergraph representation (without the use of original UML models).

Metamodels and models of UML and IM

In our running example, the transformation of fault tolerant structures will be performed from static UML models (depicted in Fig. 7.3) to this intermediate graph representation called Intermediate Model (IM in Fig. 7.4).

Example 7.12 (Extract of the UML metamodel). The simplified metamodel of UML (describing stereotyped objects and links between them with abstract classes printed in italics) is depicted in Fig. 7.3(a). The metamodel is enriched with two dependability parameters: the fault occurrence rate \( FO \) in objects and fault propagation probability \( PP \) of links (as potential propagation paths).
Example 7.13 (A sample UML model). The sample source UML model (in a visual UML notation in Fig. 7.3(b)) represents a fault-tolerant structure which consists of three objects, a redundancy manager (redundancy manager) and two variants (pressHW1 and pressHW2) identified by the corresponding stereotypes (red_man and variant) and the links between them. The redundancy manager is responsible for switching from one variant to the other when an error is detected.

Example 7.14 (The IM metamodel). The target IM metamodel in Fig. 7.4(a) specifies a hypergraph consisting of (i) nodes of type FTS representing the fault tolerant structure as a whole (ii) nodes of type Component standing for system components; (iii) and edges of type CEdge indicating the “composed of” relation.

Example 7.15 (A sample IM model). In the sample target IM hypergraph model in Fig. 7.4(b), (i) a single graph node of type Component is assigned to each variant object (var1, var2). (ii) two distinct nodes (fts1 and rml of types FTS and Component, respectively) are assigned to each redundancy manager. (iii) the fts node is in a ‘composed of’ relation with the remaining three nodes as indicated by the edges of type CEdge.

The UML2IM model transformation

Our sample model transformation is carried out by three transformation rules applied in the specific order: ftsR, variantR and linkR. The process of our sample transformation is the following (characterized by an implicit for all application of rules).
1. Transform all the redundancy managers in the UML model into two connected IM nodes (ftsR; Fig. 7.5).

2. Create a new IM node for each UML object with stereotype “variant” (variantR; Fig. 7.6).

3. Link (by applying linkR in Fig. 7.7) each IM equivalent of variant objects with the corresponding equivalent of a node of type fts (i.e. fault tolerant structure). By equivalent we mean related reference nodes and edges between the elements.

Fig. 7.5. Model transformation rule ftsR

Fig. 7.6. Model transformation rule variantR

Fig. 7.7. Model transformation rule linkR

The construction of the target IM model is illustrated in Fig. 7.8. Nodes that have been created most recently are colored grey while new edges have dashed lines.

Step 1: An fts node and an component node are created by applying the transformation rule ftsR.
Step 2: The component nodes are constructed by applying transformation rule \texttt{variantR} for the two occurrences of a source variant object.

Step 3: The \texttt{fs} nodes are connected to the component nodes (as a result of applying \texttt{linkR}) by adding a new IM edge. Each pair of these nodes are linked only once, which is ensured by the negative context condition on the LHS.

### Syntactic rules of modeling languages UML and IM

Now, another model transformation system is created to specify how to syntactically build up the source and target languages defined by graph grammar rules. Such a small grammar for handling the running example may be the following (shown in Fig. 7.9 and 7.10, where S is a placeholder for any empty LHS).

![Graph grammar of the source UML models](image)

- Let \( \mathcal{M}_8 \) be a model transformation system with source grammar \( \mathcal{A} \) of Fig. 7.9, target grammar \( \mathcal{B} \) of Fig. 7.10, and model transformation rules \( \mathcal{R}_7 \) are the ones in Fig. 7.5, 7.6, and 7.7.
- Let \( \mathcal{T} \) be a model transformation with source model \( \mathcal{M}_4 \) of Fig. 7.3(b) and model transformation rules \( \mathcal{R}_7 \).
- Let \( \mathcal{M}_5 \) of Fig. 7.4(b) be a target model candidate.

**Proposition 7.16.** \( \mathcal{T} \) is correct (with respect to \( \mathcal{B} \)).

*Proof.* It is sufficient to show a sequence of derivations that is able to construct \( \mathcal{M}_5 \) from graph grammar \( \mathcal{B} \). In other words, we have to show that the sentence \( \mathcal{M}_5 \) can be parsed by using the inverse rules of grammar \( \mathcal{B} \).
7.2 Syntactic Correctness and Completeness of Transformations

\[ S \rightarrow \text{addNode} \]

\( \text{TYPE} = \{\text{fts, component}\} \)

(a) Adding a node

\( \text{addEdge} \)

(b) Adding an IM edge

Fig. 7.10. Graph grammar of the target IM models

Let us consider the following sequence (in the given order within the same group), where \( \text{addNode}(A,B) \) means to add the node B of type A, while \( \text{addEdge}(C,D) \) adds an edge between nodes C, and D:

1. \( \text{addNode}(\text{fts}, \text{im}1), \text{addNode}(\text{component}, \text{im}2), \text{addEdge}(\text{im}1, \text{im}2) \)
2. \( \text{addNode}(\text{component}, \text{var}1), \text{addNode}(\text{component}, \text{var}2) \)
3. \( \text{addEdge}(\text{im}1, \text{var}1), \text{addEdge}(\text{im}1, \text{var}2) \),

One can easily notice that the derivation process is similar to the one of Fig. 7.8. As a result, the same graph is constructed in each case. □

**Corollary 7.17.** \( \mathcal{J} \) is complete (with respect to \( \mathcal{M}_B \)).

**Proposition 7.18.** \( \mathcal{M}_J \mathcal{S} \) is correct.

**Proof.** In this proof, each application of a model transformation rule will be related to a sequence of target grammar rules. Thus, when a model transformation rule is applied, we try to apply the corresponding sequence of grammar rules. If the target model candidate and the parallelly generated target model is isomorphic after each model transformation step then the model transformation must be correct as well.

In other words, starting from the target graph on the LHS of a model transformation rule, the target graph on the RHS of the similar rule has to be created by the graph grammar rules of the target language (empty target side is related to the \( \delta \) start symbol).

Let us consider the following coupling of rules (depicted in Fig. 7.11)

<table>
<thead>
<tr>
<th>Model transformation rules</th>
<th>Target grammar rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ftsR} )</td>
<td>\text{addNode}(\text{fts, FTS}), \text{addNode}(\text{component, RM2}), \text{addEdge}(\text{FTS, RM2})</td>
</tr>
<tr>
<td>( \text{variantR} )</td>
<td>\text{addNode}(\text{component, VAR})</td>
</tr>
<tr>
<td>( \text{linkR} )</td>
<td>\text{addEdge}(\text{FTS, VAR})</td>
</tr>
</tbody>
</table>

Fig. 7.11. Rule coupling for proving correctness

As a result, the modifications performed by model transformation rules on the target model candidate are simulated by graph grammar rules of the target language, thus \( \mathcal{M}_J \mathcal{S} \) is correct. □
Proposition 7.19. $\mathcal{M}_{T}$ is not complete (unfortunately).

Proof. For a counterexample, let us consider a target model $\mathcal{M}_{T}$ with an individual node of type $\text{fts}$. Let us suppose that there exists a source model $\mathcal{M}_{S}$ which can be transformed to $\mathcal{M}_{S}'$.

Such a source model must contain a redundancy manager object as it is the only object that is projected into an $\text{fts}$ node. When performing the transformation of the redundancy manager, an additional node and edge will appear in the target model candidate.

As the graph grammar of the target language does not contain any rules that would be able to remove graph nodes and edges, the original target model must be a subgraph of the resulting model candidate. □

Note that if another set of model transformation rules were used (splitting $\text{fts}$ into three rules: one is generating the $\text{fts}$ node and the second one derives the component node, finally, the third one creates the link between those two), completeness could have been proved.

All the proofs presented here (especially the proofs of correctness) may serve as skeletons for further proofs in connection with model transformation systems. However, the problem of an automated verification still remains. In the following, the sketch of such an automated proof method for syntactic correctness is presented, based upon planner algorithms of artificial intelligence.

7.2.4 Proving correctness by planner algorithms

Planer algorithms [193] are complex, hierarchical problem solving procedures subdividing the original problem into smaller parts before trying to solve them according to the “divide and conquer” principle. Finally, these partial solutions are merged together yielding the solution of the original problem.

Definition 7.20. A planner $\mathcal{P}A : (\mathcal{I}, \mathcal{E}, \emptyset) \rightarrow \mathcal{P}$, is a structure where $\mathcal{I}$ is the first order logic formulae of the initial state, $\mathcal{E}$ is the first order logic formulae of the goal state, while $\emptyset$ is the set of permitted operations. The output is plan $\mathcal{P}$, which is a sequence of operations providing a trajectory from the initial to the goal state.

Definition 7.21. A planner operation $\mathcal{O} = (\mathcal{C}, \mathcal{A})$, where $\mathcal{C}$ stands for the preconditions (first-order logic formulae), and $\mathcal{A}$ for actions. Preconditions must hold before performing the specific operation. Actions may add or remove certain basic logic formulae (called facts) to the state space.

In the following, a planner will be constructed to prove correctness of model transformations.

- Basic facts are built up from model graphs (supposing the close world assumption, i.e. when all the true facts have to be listed explicitly) similarly to the Prolog representation introduced in Sec. 5.2.1.
  - From a model graph node of type type with an identifier id the predicate type(id) is generated
  - From a model graph edge of type type with its own id, source src and target trg identifiers, the predicate type(id,src,trg) is generated.
  - From a model graph attribute attached to the node identified by id with a name name, and having value value, the predicate name(id,value) is generated.

- Graph grammar rules (of the source and target language) are encoded into planner operations according to the following mapping:
  - The LHS of a rule together with application conditions are encoded into a planner precondition.
    1. LHS objects are encoded into positive predicates in the Prolog style, i.e. with (unbound) variables for ids.
    2. Negative application conditions are (universally quantified) negative statements.
3. Further general conditions concerning uniqueness (and context) are added to the precondition of each operation. As our graph grammars do not contain deleting rules, postconditions are implicitly defined by the LHS objects and the additions prescribed by the RHS. This way, general postconditions such as the dangling condition need not be considered.

- the changes defined by the RHS of the rule are mapped into planner actions defining element additions (serving as new postconditions)

**Definition 7.22.** Let $\mathcal{A} = (\delta, \mathcal{R}_A)$ and $\mathcal{B} = (\delta, \mathcal{R}_B)$ to form the model transformation system $\mathcal{MTS} = (\mathcal{A}, \mathcal{R}_T, \mathcal{B})$. The proof planner $\mathcal{PA}_T$ of correctness is sequence of $\mathcal{PA}_i$ sub-planners (one assigned to each model transformation rule $\mathcal{R}_i \in \mathcal{R}_T$) which are defined as follows.

- the *initial state* of $\mathcal{PA}_i$ is defined by the left target side graph of the model transformation rule,
- a *subgoal* of $\mathcal{PA}_i$ is defined by the right target side graph of the model transformation rule
- the *operations* are defined by the graph grammar rules $\mathcal{R}_B$ of the target language.

**Proposition 7.23.** If a plan can be constructed for each $\mathcal{R}_i$ then the model transformation system $\mathcal{MTS}$ is correct.

**Proof (Sketch).** Speaking in graph transformation terms, we are aiming to prove that (i) whenever a model transformation (MT) rule is applied (to one specific match), (ii) and its effects can be simulated in general by applying a specific sequence of the target graph grammar (GG) rule on the target part of MT rule graphs, (iii) this specific sequence is applicable for the specific (isomorphic) match in the host graph thus deriving the parsing steps of the host graph from parsing just the MT rule graph. (Note the differences of rule and host graphs; graph grammar and model transformation rules)

According to our construction, performing a planner operation is identical to applying the related GG rule (without deletions). According to the assumption, there exist a sequence of GG rules that derives the right target graph (and not the image of it) from the left target graph of the MT rule. Such a sequence must not create additional graph objects (as side effects) due to the lack of deleting rules and the closed world assumption for the subgoal.

When a MT rule is applied, an isomorphic image of the initial and goal states are required to be present in the host graph. Thus, applying the same sequence of GG rules to that specific matching image, it will derive the image of the goal state (and nothing else).

Constructing a proof planner for correctness was only a demonstration. Similar planners can be built for completeness as well by slight modifications.

### 7.3 Automated Formal Verification of Model Transformations

We present an automated model-level but language independent technique [183] to formally verify the correctness of the model transformation of a specific source model into its target equivalent with respect to semantic properties based on the model checking approach of Chapter 6.

#### 7.3.1 Conceptual overview

A conceptual overview of our approach is given in Fig. 7.12 for a model transformation from an fictitious modeling language $A$ to $B$. In our demonstrating example (see Sec. 8.3), $A$ will be equal to Extended Hierarchical Automata (EHA), while Petri nets will take the role of language $B$.

- **Specification of modeling languages.** As a prerequisite for the framework, each modeling language (both $A$ and $B$) should be defined precisely using metamodeling and graph transformation.
- **Specification of model transformations.** Moreover, the $A\rightarrow B$ model transformation should be specified by a set of (non-conflicting) graph transformation rules.
Automated model generation. For any specific (but arbitrary) well-formed model instance of the source language A, we derive the corresponding target model by automatically generated transformation programs (as discussed in Chapter 5).

Generating transition systems. As the underlying semantic domain, a behaviorally equivalent transition system is generated automatically for both the source and the target model on the basis of the transformation presented in Chapter 6.

Select a semantic correctness property. We select (one or more) semantic property \( p \) in the source language A which is structurally expressible as a pattern composed of the elements of the source metamodel (and potentially, some temporal logic operators).

Model check the source model. Transition system A is model-checked automatically to prove property \( p \). This model checking process should succeed, otherwise (i) there are inconsistencies in the source model itself (a verification problem occurred), (ii) our informal requirements are not captured properly by property \( p \) (a validation problem occurred), or (iii) the formal semantics of the source language is inappropriate as a counter-example is found which should hold according to our informal expectations (another validation problem).

Transform and validate the property. We transform the property \( p \) into a property \( q \) in the target language (manually, or using the same transformation program). As a potentially erroneous model transformation might transform incorrectly the property \( p \) in to property \( q \), domain experts should validate that property \( q \) is really the target equivalent of property \( p \).

Model check the target model. Finally, transition system B is model-checked against \( q \).

- If the verification succeeds, then we conclude that the model transformation is correct with respect to property \( p \) (and \( q \)) for the specific pairs of source and target models having semantics defined by a set of graph transformation rules.
- Otherwise, property \( p \) is not preserved by the model transformation and debugging can be initiated based upon the error trace(s) retrieved by the model checker.

A natural question may easily arise after the preliminary requirements of our verification framework: if the source language itself has a precise formal semantics then why do we need further model transformations into various other mathematical formalisms.

The reason for that is two-fold: on the one hand, the verification of functional requirements for the target application is probably more efficient in the chosen target mathematical language than in our model checking approach. On the other hand, much more important is the case, when the target mathematical formalism aims to carry out a quantitative analysis (e.g., to assess dependability or QoS
parameters), therefore, the verification of the source (UML) model under design and the verification of the model transformation is complimentary.

Note that at Step 2, we only require to use graph transformation rules to specify model transformations in order to use the automatic program generation facilities of VIATRA. Our verification technique is, in fact, independent of the model transformation approach (only requires to use metamodeling and graph transformation for specifying modeling languages), therefore it is simultaneously applicable to relational model transformation approaches as well.

Naturally, the correctness of a model transformation can only be deduced if the transformation preserves every semantic correctness property used in the analysis. Obviously, it requires several runs of the model checker, which can be time-consuming. Therefore, in [172], we assessed the expected run-time performance of our model checking based approach on a verification benchmark. In [16], the same technique was applied on architectural styles to check reachability properties. Both case studies demonstrated that our technique is applicable to non-trivial examples (of medium-size).

A detailed case study aiming to carry out the formal verification of a complex model transformation is discussed in Sec. 8.3 to demonstrate the practical feasibility of our approach.

7.3.2 Metamodel vs. model level verification of model transformations

In theory, it would be advisable to prove that this model transformation preserves certain semantic properties for any well-formed model instance, but this typically requires the use of sophisticated theorem proving techniques and tools with a high verification cost. The reason for that relies in the fact that proving properties even in a highly automated theorem prover (like PVS [127] require a high-level of user guidance since the invariants derived directly from metamodels should be typically manually strengthened in order to construct the proof. In this sense, the effort (cost and time) related to the verification of a transformation would exceed the efforts of design and implementation which is acceptable only for very specific (safety-critical) applications.

However, the overall aim of model transformations is to provide and automated framework for the formal analysis of concrete applications (i.e., UML models). Therefore, in practice, it is sufficient to prove the correctness of a model transformation for any specific but arbitrary source model. Vulgarly speaking, who cares that a model transformation might be incorrect in certain hypothetical situations if it is correct for the current application?

Thanks to existing model checker tools and the transformation presented in Chapter 6, the entire verification process can be highly automated ("push-button"), Note that the selection of a pair (p,q) of corresponding semantic properties is the only part in our framework that requires user interaction and expertise. Therefore, prior to a formal analysis of the target application, the transformation designer can easily verify (at least in theory) whether certain semantic properties are preserved by the transformation.

Even if such a verification of a specific model transformation is practically infeasible due to state space explosion caused by the complexity of the target application, model checkers can act as highly automated debugging aids for the transformation supposing that relatively simply source benchmark models are available as test sets.

7.4 Conclusions and Future Work

I defined consistency criteria for model transformations and I proposed methods to formally verify the syntactic (proving language containment) and semantic consistency (aiming at property preservations) of such transformations.

• Consistency criteria for model transformations. I proposed general consistency criteria for model transformations (Sec. 7.1.1 based upon [170,188,191]) which has to be provenly satisfied by any specific model transformation.
• *Proving syntactic correctness and completeness of model transformations.* I elaborated a method based on planner algorithms (Sec. 7.2 following [187, 188, 190, 191]) for proving the syntactic correctness and completeness of model transformations. As a consequence, we may decide whether the target model (obtained as a result of a model transformation) is provenly a well-formed model of the target modeling language.

• *Semantic consistency analysis of model transformations.* Based upon the model checking technique discussed in Chapter 6, I proposed a general and automated framework (in Sec. 7.3 based on [183]) which allows to investigate whether certain (transformation specific) semantic consistency properties are preserved by a model transformation, i.e., if such a property holds in the source language, its target equivalent property will hold in the target language.

**Conceptual and practical relevance**

The main relevance of my approach is that it enables to reason about the property preservation (and other self-identified correctness criteria) of model transformations specified by a very high-level formalism. Note that the strong proof method for property preservation might make non-determinism and uniqueness checks obsolete, since it is not a problem to have various models as potential (non-deterministic) results of a transformation if all these models have a certain property.

As soon as the correctness of a model transformation is guaranteed by these techniques, we can apply the automated program generation facilities of Chapter 5 in order to derive a provenly correct implementation of a provenly correct specification. As a result, we attain transformations of a higher quality in comparison with preceding attempts in HIDE [24].

It is worth emphasizing that our model checking based approach is (up to our knowledge) the first language independent and automated solution to formally analyze the preservation of behavioral properties of model transformations.

**Feasibility of the approach**

The practical feasibility of these approaches was demonstrated on benchmark model transformations, namely, verifying the syntactic correctness of the UML2M model transformation in Sec. 7.2.3-7.2.4, and the property preservation of the EHA2PN transformation in Sec. 8.3.3.

**Limitations**

Since both planner algorithms and model checking tools have certain practical limitations due to the large complexity of IT systems, we cannot claim that we can always verify in practice that model transformations are free of conceptual flaws. However, these techniques provide a very high-level of automation to detect conceptual bugs of high-level specifications of model transformations in an early phase of transformation design.

**Future work**

Further research is primarily aiming at to provide automated projection of operational semantics of one language into another. Our idea is to adapt the semantic framework [46] based on abstract interpretation for capturing transformations of programming languages. More specifically, to apply the same model transformation rules not just to the source model but also to its operational semantics (graph transformation rules) in order to obtain an abstraction of the behavior. In this sense, entire semantics of new modeling languages could be defined and reused by model transformations, which extends rule refinement (cf. Sec. 4.5) which was our initial attempt into this direction.
Benchmarks on Modeling Languages and Model Transformations

In the current chapter, we apply our model transformation framework on various case studies. First, we formalize the dynamic operational semantics of popular visual modeling languages such as Extended Hierarchical Automata (a formal representation of UML statecharts) and Petri nets. Moreover, we discuss a model transformation from the Extended Hierarchical Automaton representation of (a meaningful subset of) UML statecharts into a Petri Net equivalent which was developed for a Hungarian research project [91]. As a result of this model transformation, we can formally analyze UML statecharts by Petri net tools. Finally, we formally verify the correctness of this model transformation with respect to the preservation of a semantic consistency property.

8.1 Case Study on Petri Nets

8.1.1 A semi-formal introduction to Petri nets

Petri nets are widely used means to formally capture the dynamic semantics of concurrent systems. However, due to their easy-to-understand visual notation and the wide range of available tools, Petri nets are also used for simulation purposes even in industrial projects (reported e.g., in [152]). From an UML point of view, transforming UML models to Petri nets provides correctness [133], dependability [26] and performance analysis [90] for the system model in early stages of design.

In order to revisit the VPM formalization of Petri nets (already discussed in Sec. 2.4.1), we present the metamodel and a sample Petri net model in Fig. 8.1.

---

**Example 8.1.** The metamodel of Petri nets in Fig. 8.1 is a slightly modified version of the one in Fig. 2.10 since we do not distinguish between tokens (thus tokens are introduced as attributes and not as classes).
A sample Petri net model is also depicted (in the middle of Fig. 8.1) in its well-known visual notation. The model contains three places $p_1, p_2, p_3$, a single transition $t_1$, a token in places $p_1$ and $p_2$, two input arcs $a_1$ and $a_2$ (leading to $t_1$ from places $p_1$ and $p_2$, respectively), and two output arcs $o_1$ and $o_2$ (leading from $t_1$ from places $p_1$ and $p_3$, respectively).

The dynamic behavior of Petri net models is captured by the notion of firing a transition.

**Definition 8.2 (Informal semantics of Petri nets).** Firing a transition in a Petri net can be defined as follows.

1. A transition is enabled when all the places with an incoming arc to the transition contain at least one token (we suppose that there is at most one arc between a transition and a place for the sake of simplicity).
2. A single transition is selected at a time from the enabled ones to be fired.
3. When firing a transition, a token is removed from each incoming place, and a token is added to each outgoing place (of a transition).
4. When no transitions are enabled, the net is dead.

Considering the Petri net model in the middle of Fig. 8.1, Petri net experts can easily conclude that initially, transition $t_1$ is enabled, since both incoming places $p_1$ and $p_2$ contain a token. As there are no additional transitions in the net, $t_1$ is immediately scheduled for firing. As a result of firing the transition, tokens are removed from places $p_1$ and $p_2$, while new tokens are generated for places $p_1$ and $p_3$ yielding the Petri net model in the right side of Fig. 8.1. Now the net is dead, as $t_1$ is no longer enabled since there are no tokens in place $p_2$.

Based upon this informal description, we can easily formalize the semantics of Petri nets in the traditional way by using abstract state machines as follows.

**Definition 8.3 (ASM semantics of Petri nets).** Let $\Sigma_{\text{pn}} = \{\text{in}/2, \text{out}/2, \text{tokens}/1, \ldots\}$ be the vocabulary of Petri nets (implicitly including the traditional function symbols and constants of integer arithmetic).

The superuniverse $|\mathfrak{A}|$ of a state $\mathfrak{A}$ in Petri nets contains the set of place identifiers $\text{Place}$ and transition identifiers $\text{Trans}$, and the set of integers $\mathbb{N}$.

The interpretation of function symbols of $\Sigma_{\text{pn}}$ is as follows.

\[
\text{in}^\mathfrak{A}(t: \text{Trans},p: \text{Place}) : \text{bool} := \begin{cases} 
\text{true}, & \text{if } p \text{ is an input place of transition } t; \\
\text{false}, & \text{otherwise.}
\end{cases}
\]

\[
\text{out}^\mathfrak{A}(t: \text{Trans},p: \text{Place}) : \text{bool} := \begin{cases} 
\text{true}, & \text{if } p \text{ is an output place of transition } t; \\
\text{false}, & \text{otherwise.}
\end{cases}
\]

\[
\text{token}^\mathfrak{A}(p: \text{Place}) : \mathbb{N} := \text{the number of tokens at place } p
\]

The semantics of Petri nets is capture the rule $\text{firePN}$ which describes how to fire a transition in the net.

**rule** $\text{firePN} =$

1. choose $T : \text{Trans}$ with $\exists P_{\text{in}} : \text{in}(T,P_{\text{in}}) \land \text{tokens}(P_{\text{in}}) = 0$
2. forall $P_{\text{in}} : \text{Place}$ with $\text{in}(T,P_{\text{in}}) \land \neg \text{out}(T,P_{\text{in}})$
3. tokens$(P_{\text{in}}) := \text{tokens}(P_{\text{in}}) - 1$
4. end for
5. forall $P_{\text{out}} : \text{Place}$ with $\text{out}(T,P_{\text{out}}) \land \neg \text{in}(T,P_{\text{out}})$
6. tokens$(P_{\text{out}}) := \text{tokens}(P_{\text{out}}) - 1$
7. end for
8. end choose
8.1.2 Semantics of Petri nets by model transformation systems

For non-experts in Petri nets, we would like to achieve that, for instance, our VIATRA model transformation framework (which is not a Petri net tool) would simulate how to fire a transition. For this reason, the semantics of Petri nets is captured by a corresponding set of graph transformation rules driven by a control flow graph as shown in Fig. 8.2. Note that this is a slightly simplified version of the original one presented in [181], which fits better to our verification purposes (see Chapter 6) as it introduces less intermediate steps.

![Fig. 8.2. Operational semantics of Petri nets](image)

As the initial step of our formalization, we need to extend the previous metamodel of Petri nets by additional features (such as the fire or tokens attributes) necessitated to capture the dynamic parts.

Theoretically, we should provide two set of rules: one for initialization and one for execution. However, in case of Petri nets, it turns out that there is no need for derived relations, thus the initialization part can be omitted (see Sec. 4.4.2 for the separation of execution phases). Therefore, the informal interpretation of the rules (given in the order of their application) is as follows.

1. First, fire attributes are set to false for each transition of the net by applying rule delFireR in forall mode.
2. A transition $T$ can be fired (by applying enableTrR) only if for all incoming arcs $A$ linked to a place $P$, this place must contain at least one token ($token>0$). Note that this time an the attribute condition
substitutes the double negation already discussed in Example 4.6. This rule is applied in `try` mode to select a single transition to be fired at a time.

3. A token is removed from all places (i.e., the counter `token` is decremented) that are connected to an `InArc`. The corresponding rule (`delTokenR`) is applied in `forall` mode.

4. For each places connected to the transition to be fired by outgoing edges `OutArc`, a token is generated (by incrementing the counter `token`) in a `forall` rule application, and thus the micro step is completed.

**Definition 8.4.** The ASM formalization of the previous semantics is presented below. Since the transformation does not manipulate the metamodel of Petri nets we may use the model-level representation of VPM models for the sake of simplicity (which is an extension of the meta-level notation thus no problems arise concerning their equivalence).

```
rule cfgPN =
  1: forall(delFireR);
  2: if try(enableTransR) then
  3:   forall(delTokenR);
  4:   forall(addTokenR)
  5: end if

rule delFireR(T, M1, M) =
  6: if transition(T) \land fire(M1) \land bool(false) \land from(M1) = T \land to(M1) = true then
  7:   delMappingFfromAtob(M1, T, true);
  8:   addMappingFfromAtob(M, T, false);
  9:   addXInstanceOfY(M, fire);
 10: end if

rule enableTransR(T, M1, M) =
  11: if transition(T) \land fire(M1) \land bool(false) \land from(M1) = T \land to(M1) = false \land
      \exists A, P, C1, C2, M2, X : inArc(A) \land toTr(C1) \land from(C1) = A \land to(C1) = T \land fromPl(C2) \land place(P) \land from(C2) = A \land to(C2) = P \land token(M2) \land integer(X) \land from(M2) = P \land to(M2) = X \land (X, 0) then
  12:   delMappingFfromAtob(M1, T, false);
  13:   addMappingFfromAtob(M, T, true);
  14:   addXInstanceOfY(M, fire);
  15: end if

rule delTokenR(T, P, A, C1, C2, M1, M2, X, Y, M) =
  16: if transition(T) \land fire(M1) \land bool(false) \land from(M1) = T \land to(M1) = true \land inArc(A) \land toTr(C1) \land from(C1) = A \land to(C1) = T \land fromPl(C2) \land place(P) \land from(C2) = A \land to(C2) = P \land token(M2) \land integer(X) \land from(M2) = P \land to(M2) = X \land (X, 0) then
  17:   Y := X - 1;
  18:   delMappingFfromAtob(M2, P, X);
  19:   addMappingFfromAtob(M, P, Y);
  20:   addXInstanceOfY(M, token);
  21: end if

rule addTokenR(T, P, A, C1, C2, M1, M2, X, Y, M) =
  22: if transition(T) \land fire(M1) \land bool(false) \land from(M1) = T \land to(M1) = true \land inArc(A) \land fromTr(C1) \land from(C1) = A \land to(C1) = T \land toPl(C2) \land place(P) \land from(C2) = A \land to(C2) = P \land token(M2) \land integer(X) \land from(M2) = P \land to(M2) \land X then
  23:   Y := X + 1;
  24:   delMappingFfromAtob(M2, P, X);
  25:   addMappingFfromAtob(M, P, Y);
  26:   addXInstanceOfY(M, token);
  27: end if
```
We can easily validate that this formalization corresponds to the semantics of Petri nets. The only crucial difference is that firing a Petri net transition (a Petri net micro step) is divided now into more elementary micro steps of the graph transformation engine, which serves as a meta-interpreter in this way.

**Proposition 8.5.** The model transition system based semantics (Fig. 8.2) is a correct formalization of the Petri net semantics of Def. 8.3. Formally, there exists a proof map \( F \) such that for all \( \mathcal{A}^m, \mathcal{B}^m, \mathcal{A}^{vpm}, \mathcal{B}^{vpm} : \mathcal{A}^m = \mathcal{F}(\mathcal{A}^{vpm}) \wedge \mathcal{B}^m = next_{\text{firePN}}(\mathcal{A}^m) \wedge \mathcal{B}^{vpm} = next_{\text{cfgPN}}(\mathcal{B}^m) \rightarrow \mathcal{B}^{vpm} = \mathcal{F}(\mathcal{B}^{vpm}) \), where rule \( \text{cfgPN} \) is the ASM rule imposed by the control structure.

Note that the proofs of propositions appearing in the current chapter are omitted from the current thesis for space considerations. All of these proofs can be carried out in a very similar way as done for Propositions 6.10–6.11.

### 8.1.3 Simulating a Petri net in the Prolog engine

Based upon the program generation techniques presented in Sec. 5.2, we may automatically derive a corresponding Prolog representation of these rules, which can be fed to the VIATRA model transformation engine.

At first, the high-level Prolog clause \( \text{cfgPN} \) is presented below, which is almost identical with the ASM representation in Def. 8.4.

```prolog
cfgPN :-
    forall(delFireR),
    try(enableTransR),
    forall(delTokenR),
    forall(addTokenR).
```

The main difference is the if structure, which is split into two separate clauses \( \text{cfgPN} \) with the same head. The second clause describes the else branch, which is now empty. In this respect, if the execution fails when calling \( \text{enableTransR} \) then we directly backtrack to the consecutive \( \text{cfgPN} \) clause head since no choice points are generated for the Prolog implementation of the forall construct.

The Prolog representation of \( \text{delFireR} \) and \( \text{enableTransR} \) is rather self-explanatory after Sec. 5.2 and Def. 8.4, the reader might notice that attribute conditions \( x == 0 \) are translated into built-in Prolog tests.

```prolog
delFireR :-
    % LHS
    node(pn:transition(T)),
    attr(pn:fire(A0, T, true)),
    % RHS
    remove(attr(pn:fire(A0, T, true))),
    add(attr(pn:fire(_A1, T, false))).
```

```prolog
enableTransR :-
    % LHS
    node(pn:transition(T)),
    attr(pn:fire(A0, T, false)),
    % NEG
    (
```

edge(pn:toTr(E0, A, T)),
node1(pn:inArc(A)),
edge(pn:fromP1(E1, A, P)),
node1(pn:place(P)),
attr(pn:token(A1, P, X)),
X == 0 -> % attribute condition
fail
true
),
% RHS
remove(attr(pn:fire(A0, T, false))),
add(attr(pn:fire(_, A2, T, true))).

However, further considerations are need to handle delTokenR, and especially, addTokenR. The
first problem is to map the token-- (token++) statements into a corresponding Prolog equivalent.
This can be done by introducing a new variable X1 and an assignment X1 is \( X-1 \) (X1 is \( X+1 \)) to
avoid the problem that variables cannot be refreshed in Prolog. A second problem for addTokenR is
that the token attribute has to be queried explicitly in order to obtain the old value to be incremented.
This was not part of the corresponding ASM description where variables (nullary functions) can be
updated without explicitly accessing its former value prior to the update operation.

delTokenR :-
% LHS
node(pn:transition(T)),
attr(pn:fire(A0, T, false)),
edge(pn:toTr(E0, A, T)),
node1(pn:inArc(A)),
edge(pn:fromP1(E1, A, P)),
node1(pn:place(P)),
attr(pn:token(A1, P, X)),
X > 0, % attribute condition
% RHS
X1 is X-1, % variable assignment
remove(attr(pn:token(A1, P, X))),
add(attr(pn:token(_, A2, P, X1))).

addTokenR :-
% LHS
node(pn:transition(T)),
attr(pn:fire(A0, T, false)),
edge(pn:toTr(E0, A, T)),
node1(pn:inArc(A)),
edge(pn:fromP1(E1, A, P)),
node1(pn:place(P)),
attr(pn:token(A1, P, X)), % obtaining the old value of token
% RHS
X1 is X+1,
remove(attr(pn:token(A1, P, X))),
add(attr(pn:token(_, A2, P, X1))).
8.1.4 Rule refinement in defining semantic variations of Petri nets

For a more complex example of using rule refinement (of Sec. 4.5), let us consider that we want to derive from the general Petri net metamodel the specific metamodel of 1-bounded or 1-safe Petri nets where each place may only contain at most one token at a time. Therefore, some slight changes are needed to be introduced in the Petri net semantics to capture that a transition is only enabled if all incoming places contain a token but none of the outgoing places (that are not incoming places as well) contains a token.\(^1\)

Our goal is to maintain as much as possible from the original Petri net formalization in Fig. 8.2, and introduce a new rule (Fig. 8.3) by rule inheritance that modifies the semantics in the right way. As a result, the dynamic operational semantics of a language will also be reused when defining a more refined language (in addition to reusing its structural definition).

![Diagram of 1-safe Petri net](image)

**Fig. 8.3.** 1-safe Petri nets: modifying semantics by rule refinement

- As for structural changes in the metamodel of 1-safe Petri nets, the tokens connection is refined into a tokens mapping (as indicated by the highlighted areas).
- In the dynamic parts, the only new rule is `safeEnableR`, which is a proper refinement of the former version of the rule `enableTransR`, since a new negative condition has been introduced that prescribes that no outgoing places (see place `P1`) are permitted to contain a token `M` prior to enabling the transition `T`, except for those that are also serve as incoming places for the same transition (connected by `InArcs` as well). One can easily check that the rule refinement relation holds between rules `enableTransR` and `safeEnableR` by the following argument.
  - \(Lhs_{safeEnabR} \subseteq Lhs_{safeEnableR}\), since they are identical (and the refinement of `tokens` from a connection to a mapping is valid).

\(^1\) From a mathematical point of view, this subclass of Petri nets can be simulated by the original semantics as well with a structural modification that introduces a complementary place for each existing one.
- $Neg_{sa} \subseteq Neg_{sa\text{Trans}}$, since an additional negative condition has been introduced while all existing conditions are left unaltered.
- Conditions 3 and 4 trivially hold as the right-hand sides of rules are identical.

As a result, we enabled the reuse of dynamic operational semantics of a modeling language by providing a precise means of refining graph transformation rules in addition to the conventional reuse of structural elements.

### 8.2 Extended Hierarchical Automata: Formal Semantics of UML Statecharts

Extended Hierarchical Automata (EHA) serve as a common semantic basis for defining formal semantics of UML statecharts in an operational way. In the current section, we use only the structure of the same formalism; however, the operational semantics will be defined by means of model transition systems. Again, we validate that our semantics is a equivalent (up to observables) with the traditional EHA semantics of \cite{105}.

At first, we very briefly introduce UML statecharts (based upon \cite{105,168,174,178}).

#### 8.2.1 An informal overview of UML Statecharts

UML statecharts (see the example in Fig. 8.4) are an object-oriented variant of classical Harel statecharts \cite{79} that describe behavioral aspects of a class in the system under design. In fact, the statechart formalism itself is an extension of finite state machines (FSMs) to allow a decomposition of states into a state hierarchy that greatly enhance the readability and scalability of state models.

![Fig. 8.4. A sample UML statemachine](image)

UML Statemachines are basically constructed from states (including the top state of the hierarchy) and transitions.

**States**

According to state refinement rules in UML statecharts, states can be either simple states, OR-states (non-concurrent) or AND-states (concurrent) (disregarding from pseudo states like initial and final states).

Simple states (like $s_2$ or $s_7$ in Fig. 8.4) are at the final level of refinement. State $s_1$, on the contrary, is refined into two distinct regions (represented as a state in UML), $s_4$ and $s_5$, each of them is refined in turn into an automaton consisting of further substates (e.g. $s_6$, $s_7$).

States refined to sub-states are denoted as composite, additionally, $s_4$ and $s_5$ are called concurrent regions of the concurrent state $s_1$ as each of them has an active substate. As a general semantic...
well-formedness rule, an active OR-state may have exactly one active substate, while an active AND-state has an active substate in each of its subregions.

At one point in time, the set of all active states forms the active configurations. For instance, our sample system can be any of the following configurations: \{s1.s6.s8\}, \{s1.s6.s9\}, \{s1.s7.s8\}, \{s1.s7.s9\}, \{s2\}, \{s3\}.

Transitions

Transitions show how objects change state. A Transition connects a source state to a target state. A transition is labeled by a trigger event, a boolean guard and a sequence of actions.

A transition is enabled (and may fire later on) if and only if its source state is in the current configuration, its trigger is offered by the external environment (from a message queue) and the guard is satisfied.

As a result, more than one transition can be enabled, which may cause a conflict if the intersection of the states left by the enabled transitions is not empty.

Conflicting transitions are tried to be resolved by using priorities: a transition has higher priority than another transition if its source state is a substate of the other transition’s source state. A transition having sufficient priority (i.e., not disabled when resolving conflicts) is called fireable.

If conflicts cannot be resolved by priorities, any of the fireable transitions can be fired, moreover, according the run-to-completion step, all transitions from the non-conflicting subset of fireable transitions are fired at a time. When firing a transition, the source state is left, the actions are executed, and the target state is entered.

In our example, if event e1 is offered by the environment and the current configuration is \{s2\}, then state s2 is left and state s1 is entered. In particular, as e1 is composite, we also have to define which are the substates that are reached. In the case at hand, they are the default ones specified by the initial states of s4 and s5, namely, s6 and s8. In a general case, the source and target state of a transition may be at a different level of the state hierarchy. Such a transition is denoted then as interlevel.

Event dispatching

In general, more than one event can be available in the environment (message queue). The UML semantics assumes a dispatcher, which selects one event at a time from the environment and offers it to the state machine using a strategy left undefined by the UML standard (e.g., may be a FIFO or multiset).

Non-standard UML extensions

In the current version of our statechart semantics, several concepts of the UML standard (such as history states, deferred events) have been omitted for space limitations. We hope that our formalization concepts in Section 8.2.4 will demonstrate that the “neglected” parts can easily be integrated if required.

On the other hand, we also had to extend the original UML standard due to the lack of a proper specification for event queues. Currently, one queue is attached separately to each object and it may store a single instance of all valid events arriving to the objects. However, this approach can easily be extended to alternate queue models (FIFO, multiset, etc).

Modeling restrictions

Unfortunately, UML does not specify precisely enough all the consistency constraints between different UML diagrams. Therefore different local views of the system (as provided from different aspects by various UML diagrams) cannot be integrated into a consistent and semantically well-founded global view in a straightforward way.
Therefore, we now impose certain restrictions on UML models that ensure consistency for the combination of class, object and statechart diagrams. Note that these are (unfortunately) not part of the standard (therefore a UML CASE tool does not enforce the systems designers to respect them), however, violating them would prevent the designers from using the presented Petri nets based formal analysis technique. As additional restrictions on UML models are frequently imposed by the use of specific UML profiles for modeling in certain application domains, we believe that our restrictions are also meaningful for obtaining a more consistent design methodology for UML.

Example 8.6 (Voting). Our UML restrictions are introduced on a simple UML design (shown in Fig. 8.5) modeling a voting process which requires a consensus (i.e., unanimous decision) from the participants.

![Fig. 8.5. UML model of unanimous voting](image)

In the system, a specific task is carried out by multiple calculation units CalcUnit, and they send their local decision to the Voter in the form of a yes or no message. The voter may only accept the result of the calculation if all processing units voted for yes. After the final decision of the voter, all calculation units are notified by an accept or a decline message. In the concrete system, two calculation units are working on the desired task (see the object diagram in the upper right corner of Fig. 8.5), therefore the statechart of the voter is rather simplified in contrast to a parameterized case.

Moreover, we also present in Fig. 8.6 the EHA equivalent of the voter system (of Fig. 8.5) obtained as a result of the SC2EHA model transformation discussed in [178].

![Fig. 8.6. The EHA equivalent of the UML model of voting](image)
8.2 Extended Hierarchical Automata: Formal Semantics of UML Statecharts

The main conceptual problem in the interaction of statechart, object and class diagrams is that a statemachine is assigned to a class and not to an object. Therefore, the instances of this class in the statemachine cannot be accessed in design time when addressed message sending is aimed to be specified.

We propose the use of role names of associations to specify the target objects of a message. More specifically, all instances of that are visible via the specified role name will receive such a message. For instance, the message theCalcUnit.decline implies that it will be sent at run-time to both CalcUnit objects c2 and c2 since both of them are visible from the Voter v via the links with the theCalcUnit role name. If required, we can specify with multiplicity constraints that a message is only allowed to be sent to a single object, otherwise message sending have a broadcast nature by default.

8.2.2 The metamodel of UML Statecharts

In the following, the major features and components of UML statecharts will be described by means of (an extract of) the standard UML metamodel (Figure 8.7). However, additional semantic restrictions expressed in OCL are omitted for the sake of simplicity.

- The top-level class is called Model Element. It is an abstract superclass (thus without instances) with a single attribute name.
- A state machine is a behavior that specifies the sequences of states that an object goes through during its life in response to events, together with its response and actions.

In the metamodel a StateMachine is composed of a top State and an arbitrary number of transitions.

- The top association defines the top level State (exactly one as depicted by its multiplicity 1) directly owned by StateMachine. Further States are owned by parent composite states and discussed later.
- The transitions role relate the StateMachine to its Transitions. All Transitions are owned directly by at most one StateMachine.

- A state vertex is an abstract class in the statechart. In general, it can be the source or destination of any number of transitions.

In the metamodel a StateVertex is a subclass of ModelElement.
A **state** is a condition or situation during the life of an object meanwhile it satisfies some condition, performs some action or waits for some events. In the metamodel **State** is an abstract class and a subclass of **StateVertex**.

- A **simple state** is a state that does not have substates. In the metamodel a **SimpleState** is a subclass of **State** and it does not have any additional features.
- A **composite state** is a state that consists of substates. In the metamodel a **CompositeState** is a subclass of **State**.
  - Its **subvertex** association denotes a set of **States** that form the substates of a **CompositeState**. Each substate is uniquely owned by its parent **CompositeState**, and self-containment is not allowed either.
  - The **isConcurrent** attribute has a boolean value that specifies the decomposition semantics: if this attribute is true, then the composite state is an **OR-state** thus it is decomposed directly into two or more orthogonal conjunctive regions (usually associated with concurrent execution). Otherwise, there are no direct orthogonal region in the composite state (in this case, it is called an **AND-state**). This means that exactly one of the substates can be active at a given instant (i.e. sequential execution).

- A **pseudo state** is an abstraction of different types of nodes including initial and final states. In the metamodel a **PseudoState** is a subclass of **StateVertex**. It possess the **kind** attribute that can be e.g. **initial**, **final**, **fork** or **branch**. An additional semantic constraint here should state that each non-concurrent composite state must have exactly one initial pseudo state.
- A **transition** is a binary relationship between a **source** state vertex and a **target** state vertex. In the metamodel **Transition** is a subclass of **ModelElement** that participates in various relationships with other state machine metaclasses (by associations):
  - **trigger** specifies the single **Event** which activates it
  - **guard** is a predicate that must evaluate to true at the instant the transition is triggered; for the current case study, we only consider the conjunction of **slm** expressions as guards, which prescribes that a distant state in the automaton to be active.
  - **effect** specifies an **Action** which has to be performed after the transition is fired.
  - **source** denotes the **StateVertex** affected by firing the **Transition**.
  - **target** denotes the **StateVertex** that results from a firing of the **Transition** when the **StateMachine** was originally in the source **State**. After the firing the **StateMachine** is in the target **State**.

- An **event** may lead to the activation of a some internal behavior in an object. In the metamodel an **Event** is a subclass of **ModelElement** and is a part of a **Transition** by representing its **trigger**.

- A **guard** condition is a boolean expression that may be attached to a transition in order to determine whether that transition is **enabled** or not. In the metamodel **Guard** is a **ModelElement**. Its **expression** attribute is a boolean expression which specifies the guard condition.
- In general, an **action** may also lead to the activation of a some internal behavior in an object after a transition has been fired. As a restriction for the current case study, all actions are considered to be send actions, i.e., they send a message to a target object accessible via a specific role. In the metamodel an **Event** is a subclass of **ModelElement** and is a part of a **Transition** by representing its **effect**.

Note that the UML metamodel was slightly simplified (with respect to the standard UML metamodel) in order to improve legibility of the thesis. These modifications do not have major impact on statechart semantics.
8.2.3 Related work on formal semantics for UML statechart

Prior to discussing our own formalization, we provide a brief overview (following [168]) on related approaches that define formal semantics to UML statecharts with a special emphasis on transformation based approaches followed by a formal analysis.

Since the original formalism of Harel [79], the theory of statecharts has been under an extensive research and many different semantic approaches evolved from the academic world (a comparison of different approaches can be found in e.g. [192]). However, as the industrial interest is rather limited to the Statemat and UML variants, therefore, the majority of recent approaches for statecharts semantics have typically focused on the formalization of those variants. In this section, we restrict our attention to compare only proposals for the UML dialect with a stress on the support of formal verification.

Extended Hierarchical Automata, which form the structural basis of our statechart semantics, were introduced in [111] for Statemat and in [105] for UML. In a second phase, both approaches transform their models into Promela code and verify them by the model checker SPIN [87]. A major stress is put on formal verification in [161] where UML statecharts are encoded into a PVS [128] specification enabling the access to automated theorem proving of UML design, while in [103], the model checking of UML statecharts is aimed.

An entire verification round-trip is reported in [43] and [131] where the results of model checking are represented visually in the original UML models. The semantic core of statecharts is formalized in those papers by means of abstract state machines and state term graphs, respectively.

Model transformations in the RIVIERA framework [144] also aim at to carry out formal analysis and verification of UML models by transforming them into the Maude language. In [200], state space reduction techniques are applied to UML with Action Semantics for model checking purposes.

An integrated framework called MetaEnv is reported in [15, 19] which is based on transforming visual diagrammatic notations into High-Level Timed Petri Nets as the underlying semantic domain by graph transformation rules. Their techniques are applied directly to UML in [18].

Graph transformation as UML semantics

Despite their success from a verification point of view, the use of precise, formal mathematics is also the common weakness of all these approaches: they fail to provide a high level of abstraction that can be properly understood (and implemented) by systems engineers. Previous proposals in the field of graph transformation (e.g., [73, 101]) have tried to tackle this problem by providing a visual specification of statechart semantics.

Even though these proposals derive their internal graph representation for UML models directly or indirectly from the standard UML metamodel, semantic concepts are typically hard coded into the semantic rules, which does not scale up well for different statechart languages or the future evolution of the UML language itself. For instance, to implement the inverse priority concepts of Statemat semantics would require a major revision in all these approaches.

In the current chapter, we define the dynamic behavior of UML statecharts by combining metamodelling and graph transformation techniques. However, our main contribution is to simultaneously include the purely syntactic (states, transitions, events) and derived static semantic concepts of statecharts (like conflicts, priorities, etc.) in the metamodel, but separate them from their dynamic operational semantics, which is specified by graph transformation rules. This philosophy keeps the metamodel and the graph transformation rules easy to be understood and maintained for statechart variants. Additionally, our methodology provides direct access to the formal verification of UML statecharts by applying the techniques investigated in [171, 172] (and in Chapter 6).
8.2.4 Extended Hierarchical Automata: A semantic domain of UML Statecharts

The UML version of Extended Hierarchical Automata (EHA) were introduced in [105] to provide an alternate representation and a formal operational semantics for Statechart diagrams by a small number of complex transition rules. In the current thesis, the EHA model is considered to be only an alternate structural representation of statecharts, while the original structured operational semantic (SOS) rules are replaced by a set of dynamic attributes and relations manipulated by graph transformation rules.

Note that the formalization method to be presented below could be applied directly to the language of UML statecharts. We believe that the intermediate EHA representation provides greater flexibility when further statechart variants (e.g. the Statemate or Matlab dialects) are considered in the future. In fact, the EHA notation can be derived automatically from the original UML statechart notation by the model transformation process described in [178].

**Static structure of Extended Hierarchical Automata**

The structural basis of Extended Hierarchical Automata are defined by its metamodel in Fig. 8.8. The classes of the EHA metamodel are prefixed with the letter 'h' in order to avoid name clashes with the original notions of UML statecharts.

- An Extended Hierarchical Automaton (EHA) is composed of a top `hState`, and sequential automata of class `hAut`. Additionally, an EHA is attached to an event queue `eventQ`. Each instance (in the object diagram) of a UML class that is associated with a stateemachine is projected into a distinct EHA instance.
- A sequential automaton `hAut` is generated for (i) each non-concurrent composite state and (ii) for each regions of a concurrent composite state in a UML statechart. It is composed of `hStates` (referred by `autStates`) and `hTransitions` (accessed by `autTrans`).
- Each UML state that is not a region of a concurrent state is transformed into a `hState`. The state refinement relation is preserved by the `refined` association linking states to their subautomata.
- A transition in UML statecharts has a corresponding transition `hTrans` in its EHA equivalent linking the `from` `hState` and the `to` `hState`.
  - All EHA transitions are **non-interlevel** thus connecting states that belong to the same sequential automaton, which is the EHA equivalent of the least common ancestor state in the UML stateemachine. The least common ancestor of a UML transition is the lowest level state that is a superstate of both the source and the target states.
  - The original source state(s) of transition is denoted by the **source restriction** relation (`srcRest`).

![Fig. 8.8. The EHA metamodel](image-url)

<refined>hState</refined>
<refined>hTrans</refined>
<refined>hGuard</refined>
<refined>hEvent</refined>
<refined>hAut</refined>
<refined>EHA</refined>
<refined>hQueue</refined>
<refined>hAction</refined>
<refined>eventQ</refined>
<refined>receiver</refined>
<refined>guard</refined>
<refined>validEv</refined>
<refined>effect</refined>
<refined>event</refined>
<refined>trigger</refined>
<refined>isin</refined>
<refined>substates</refined>
<refined>hState</refined>
<refined>hAct</refined>
<refined>enable</refined>
<refined>fireable</refined>
<refined>conflict</refined>
<refined>exitSt</refined>
<refined>enterSt</refined>
<refined>givePrior</refined>

<autState>
<autTrans>
<top>
<hAut>
<hState>
<hTrans>
<hGuard>
<hEvent>
<hQueue>
<hAction>
<eventQ>
<receiver>
<guard>
<validEv>
<effect>
<event>
<trigger>
<isin>
<refined>srcRest</refined>
<top>
<hAut>
<hState>
<hTrans>
<hGuard>
<hEvent>
<hQueue>
<hAction>
<eventQ>
<receiver>
<guard>
<validEv>
<effect>
<event>
<trigger>
<isin>
<refined>srcRest</refined>
The target determinator (`trgDet`) of a transition relates the set of hStates that (i) have a simple state equivalent in the original UML model, and (ii) they must be entered implicitly when the transition is fired (thus more than a single target determinator state may be connected to e.g., transition `t2`).

The source restriction and target determinator set of a transition can be generated according to the normalization algorithm presented in [106].

- A hTransition is triggered by a related hEvent, and its effect is defined by its corresponding hAction. In the paper, we restrict our attention to send actions, i.e., all the actions are considered as sending an event to the specified eventQ. The guard condition is a boolean expression (composed of `isin` conditions) that must hold to allow the transition to be enabled.
- An EHA object is automatically associated to one event queue where the messages sent to the object arrive at. Each queue stores which events are valid (`validEv`), i.e., the event is accepted by at least one transition in the state machine.

The EHA encoding of the sample statechart (of Fig. 8.4) is shown in Fig. 8.9. In order to improve the clarity of the graphical parts, we represented the events (EV), actions (AC), source restrictions (SR) and target determinators (TD) of a transition in a table. Note that grey areas in the table represent the corresponding values for `interlevel` UML transitions.

![Fig. 8.9. Notational guide: a sample EHA model](image)

**Static and dynamic EHA concepts**

In the metamodel, core static parts of the metamodel (left from the vertical line) were kept separated from dynamic (and derived) relations and attributes (right from the vertical line). All the previous concepts are regarded as static parameters, since they are derived from the original UML model at compile time. For describing the dynamic behavior of statecharts in an easy-to-understand way, additional attributes and relations are required.

- The boolean attribute `isAc` of a hState will be set to true whenever the hState is active (i.e. member of the current configuration). The attribute `evSel` of a hEvent is true when this hEvent is selected by the dispatcher (as the dispatcher belongs to the environment, this attribute will be handled non-deterministically).
- The relation `substates` connects a hState to its descendant hStates in the EHA state hierarchy. When a hQueue is related to a hEvent by an `inQueue` edge, this fact denotes that the hEvent is a member of the hQueue set.
The attributes enabled, firable and fire will denote, respectively, (i) when a transition is triggered by the selected event and its source state is an active state, (ii) it is enabled and has sufficient priority to be fired, and (iii) it is selected to be fired.

There are four additional relations for hTransitions, which are, in fact, static but not part of the EHA model introduced in [105]. The relation exitS explicitly enumerates all the states that have to be exited when the transition is fired. Similarly, the enterS relation lists all the states to be entered when a transition is fired. Two hTransitions are in a conflict relation (i.e. they might be in conflict when firing them) if their exitS set is not disjoint. While givePrior specifies the priority relation between hTransitions.

As a reminder, the major distinction between static and dynamic concepts is that static parts are not modified while an EHA is being operated in its life-cycle by the upcoming graph transformation rules. Therefore, the dynamic attributes (and relations) are initialized and not compiled. With this respect, the semantic formalization of EHA will be divided into two subsequent phases, namely, (i) an initialization (preprocessing) phase for generating derived properties and initializing dynamic constructs, and (ii) the execution of core EHA semantics itself. This distinction between dynamic and static (either derived or static) parts will be of immense importance later on in Chapter 6, where we will aim at model checking these high-level specifications.

8.2.5 An operational semantics of Extended Hierarchical Automata

Now we define the semantics of Extended Hierarchical Automaton by a model transition system. The initial graph is the static model generated by the SC2EHA transformation [178]. The top-level ehaSemantics module thus consists of two Call nodes — one for the initialization phase, and one for the operational phase.

The preprocessing phase

The intDynamics module is mainly responsible for deriving those static relations that are required for an easy-to-understand formalization of dynamic behavior. In addition, the initialization of dynamic attributes also takes place at this stage. The preprocessing phase consists of 10 rules (see Fig. 8.10) that are executed in according to the control flow graph in the following order. An alternate solution for this preprocessing phase is to use path expressions for such derived relationships.

1. substatesR1: The rules substatesR1 and substatesR2 build up the substate relationship in two steps. At first, if a hState $S_1$ is refined to a hAutomaton $A$, all the hStates $S_2$ of this automaton are substates of $S_1$ (a forall execution).
2. substatesR2: Secondly, the transitive closure of the substates relation is calculated by looping rule substatesR2 as long as possible. This means to add substates edges between hStates $S_1$ and $S_3$, if $S_2$ is substate of $S_1$, $S_3$ is a substate of $S_2$ but no substate edges are leading yet between $S_1$ and $S_3$.
3. exitStateR1: Rules exitStateR1 and exitStateR2 explicitly connect the hStates that are exited by a hTransition when the transition is fired to the hTransition. At first, the from hState $S$ of a hTransition $T$ must be exited.
4. exitStateR2: Afterwards, all the substates $S_2$ of the from state $S_1$ of the hTransition $T$ must also be exited.
5. enterStateR1: All the states target determinant hStates $S$ of a firing hTransition $T$ are should be entered when firing this transition.
6. enterStateR2: Additionally, all the states $S_2$ that are superstates of the target determinant state $S_1$ of a hTransition $T$ but not superstates of the to hState $S_3$ must also be entered. Note that the states to be exited and entered when firing a transition are static information.
7. conflictR: The rule conflictR connects two hTransitions $T_1$ and $T_2$, if there exists a hState $S$ that is linked to both of them by an exitS edge, thus it is exited by both transitions causing a conflict.
8. **givePriorityR**: According to this rule, a hTransition \( T_2 \) has lower priority than hTransition \( T_1 \) if for the corresponding source restriction hStates (\( S_2 \) and \( S_1 \), respectively), \( S_1 \) is a substate of \( S_2 \).

9. **initTransR** and **initActiveR**: Initially, all the dynamic attributes of hTransitions are set to false.

10. **initActiveR**: A state \( S \) becomes active if it is an initial state of some sequential hAutomaton \( A \).

**Example 8.7.** By the end of the preprocessing phase of the EHA model in Fig. 8.9, we derived, for instance, that

- \( s_6, s_7, s_8 \) and \( s_9 \) are substates of \( s_1 \);
- the states to be exited when firing transition \( t_1 \) are \( s_1 \) (the from state), \( s_6, s_7, s_8 \) and \( s_9 \) (the substates of the from state);
- the states to be entered when firing transition \( t_2 \) are \( s_6, s_8 \) (the target determinators) and \( s_1 \) (which is the to state);
• transitions $t_1, t_3$ and $t_6$ are in conflict with each other since state $s_6$ is exited by all of them;
• however, $t_3$ gives priority to $t_1$ and $t_6$ as the source restriction state $s_1$ of $t_1$ is a superstate of $s_6$ (the source restriction of $t_1$ and $t_6$).

**Operational phase**

Now we continue with the discussion of the “more semantical” operational phase (depicted in Fig. 8.11), where the run-to-completion step of statecharts are refined into a sequence of more elementary operations.

![Diagram](image)

**Fig. 8.11.** The model transition system specifying the EHA semantics
1. **selectEventR**: At first, an event \( E \) is non-deterministically selected from an event queue \( Q \). If no such events are available, then the execution of EHA terminates. Note, that different (more complex) event handling mechanisms can be applied at this point, however, this non-deterministic selection overapproximates the semantics of such mechanisms. In fact, transitions without trigger events (that may be fired immediately) can be modeled, for instance, with a selection method with priorities.

2. **enabledR**: A hTransition \( T \) is **enabled** if its source restriction hState \( S \) is active, its trigger hEvent \( E \) is selected by the dispatcher, and no states accessed by the *isn* guard are inactive (note that this condition also holds for transitions without *isn* guards).

3. **fireableR**: An enabled hTransition \( T \) becomes **fireable** if there are no enabled hTransitions \( T_2 \) of higher priority (see the negative condition).

4. **fireFirstR**: The first (fireable) hTransition \( T \) is selected to be fired non-deterministically by setting its *fire* attribute to true. If no such transitions found then the execution continues by resetting dynamic attributes of transitions (see the final step).

5. **fireNextR**: After the success of **fireFirstR**, the set of transitions to be fired fired is extended one by one (by looping **fireNextR**) until all the remaining enabled hTransitions are in conflict with at least one element in the fire set.

6. **exitR**: All the states \( S \) marked by an exitSt edge leading from a firing hTransition \( T \) are exited.

7. **addQueueR**: As the effect of firing a hTransition \( T \), the hEvent \( E \) associated to the (send) hAction \( A \) is added to the corresponding hQueue \( Q \) (if the negative condition is removed, then the event queue is modeled as a bag and not a set).

8. **enterR**: All the states \( S \) marked by an enterSt edge leading from a firing hTransition \( T \) are entered.

   This step results in a valid configuration as (i) the origins of target determinator states were simple states, hence no states need to be entered at a lower level than the target determinators and (ii) all EHA transitions are non-interlevel thus no states had been exited at a higher level than the *from* and to hStates.

9. **resetR** and **deselectEvR**: All the dynamic attributes of a transition \( T \) and an event \( E \) are set to false, and a new step of the EHA commences.

**Example 8.8.** Let us assume that the active states of our sample EHA model are the initial states (thus \( s_1, s_8 \), and \( s_9 \)) and the event queue only contains the single event of \( e_1 \). According to the previous rules, a run-to-completion step of our hierarchical automaton proceeds as follows.

1. Transitions \( t_3 \) and \( t_6 \) are enabled by applying **enabledR** as the source restriction states (\( s_1 \) and \( s_6 \), respectively) of both transitions are active.

2. From this enabled set of transitions, the application of **fireableR** eliminates \( t_3 \) since \( t_3 \) gives priority to \( t_6 \).

3. The set of transitions to be fired will consist of the single transition \( t_6 \).

4. By applying **exitR**, states \( s_1, s_6 \) and \( s_8 \) become inactive, while the application of **enterR** results in the activation of state \( s_2 \). Meanwhile, event \( f_1 \) is added to the event queue by rule **addQueueR**.

5. Finally, all dynamic attributes of all transitions are set to false, and a new run-to-completion step commences.

**Definition 8.9 (ASM representation of EHA semantics).** The ASM formalization of the previous semantics (Fig. 8.11) is presented below. We use use the the model-level representation of VPM models this time as well for the sake of simplicity.

**rule** fireEhaRef =

1. **if** try(selectEventR) **then**
2. forall(enabledR);
3. forall(fireableR);
4. **if** try(fireFirstR) **then**
loop(fire NextR);
forall(exitR)
forall(addQueueR)
forall(entryR)
end if
forall(resetR);
forall(deselectEvR);
end if

rule enabledR(T, S, E, CTS, CTE, MT, MS, ME, FT) =
if hTrans(T) ∧ bool(true) ∧ bool(false) ∧ enable(MT) ∧ from(MT) = T ∧ to(MT) = false ∧ hState(S) ∧ isAct(MS) ∧ from(MS) = S ∧ to(MS) = true ∧ srcRest(CTS) ∧ from(CTS) = T ∧ to(CTS) = S ∧ hEvent(E) ∧ evSel(ME) ∧ from(ME) = E ∧ to(MS) = true ∧ trigger(CTE) ∧ from(CTE) = T ∧ to(CTE) = E ∧ S1, C, M : hState(S1) ∧ isAct(M) ∧ from(M) = S1 ∧ to(M) = false ∧ isln(C) ∧ from(C) = T ∧ to(C) = S1 then
delMappingFfromAtoB(MT, T1, false);
addMappingFfromAtoB(FT, T1, true);
addInstanceOfY(FT, enable);
end if

rule fireableR(T, M1, M2, F2) =
if hTrans(T) ∧ bool(true) ∧ bool(false) ∧ enable(M1) ∧ from(M1) = T1 ∧ to(M1) = true ∧ fireable(M2) ∧ from(M2) = T2 ∧ to(M2) = false then
delMappingFfromAtoB(M2, T1, false);
addMappingFfromAtoB(F2, T1, true);
addInstanceOfY(F2, fireable);
end if

rule fireFirstR(T, M1, M2, F2) =
if hTrans(T) ∧ bool(true) ∧ bool(false) ∧ fireable(M1) ∧ from(M1) = T ∧ to(M1) = true ∧ fire(M2) ∧ from(M2) = T ∧ to(M2) = false then
delMappingFfromAtoB(M2, T, false);
addMappingFfromAtoB(F2, T, true);
addInstanceOfY(F2, fire);
end if

rule fireNextR(T, M1, M2, F2) =
if hTrans(T) ∧ bool(true) ∧ bool(false) ∧ fireable(M1) ∧ from(M1) = T1 ∧ to(M1) = true ∧ fire(M2) ∧ from(M2) = T2 ∧ to(M2) = false ∧ hTrans(T) ∧ fire(M) ∧ from(M) = T1 ∧ to(M) = true ∧ conflict(C) ∧ from(C) = T1 ∧ to(C) = T2 then
delMappingFfromAtoB(M2, T1, false);
addMappingFfromAtoB(F2, T1, true);
addInstanceOfY(F2, fire);
end if

rule exitR(T, S, CTS, MT, MS, M) =
if hTrans(T) ∧ hState(S) ∧ exitSt(CTS) ∧ from(CTS) = T ∧ to(CTS) = S ∧ bool(true) ∧ bool(false) ∧ fire(MT) ∧ from(MT) = T ∧ to(MT) = true ∧ isAct(MS) ∧ from(MS) = S ∧ to(MS) = true then
delMappingFfromAtoB(MS, S, true);
addMappingFfromAtoB(M, S, false);
addInstanceOfY(M, isAct);
end if
8.2 Extended Hierarchical Automata: Formal Semantics of UML Statecharts 199

rule enterR(T, S, CTS, MT, MS, M) =
38:  if hTrans(T) ∧ hState(S) ∧ enterSt(CTS) ∧ from(CTS) = T ∧ to(CTS) = S ∧ bool(true) ∧ bool(false) ∧
    fire(MT) ∧ from(MT) = T ∧ to(MT) = true ∧ isAct(MS) ∧ from(MS) = S ∧ to(MS) = false then
39:    delMappingFrmAtob(MS, S, false);
40:    addMappingFrmAtob(M, S, true);
41:    addInstanceOfY(M, isAct);
42:    end if

rule addQueueR(T, A, Q, E, CTA, CAF, C, E) =
43:  if hTrans(T) ∧ hAction(A) ∧ effect(CTA) ∧ from(CTA) = T ∧ to(CTA) = Q ∧ hQueue(Q) ∧
    receiver(CAF) ∧ from(CAF) = A ∧ to(CAF) = Q ∧ hEvent(E) ∧ event(CAF) ∧ from(CAF) =
    A ∧ to(CAF) = E ∧ ∃CQE : inQueue(CQE) ∧ from(CQE) = Q ∧ to(CQE) = E then
44:    addConnectionRfrmAtob(C, Q, E);
45:    addInstanceOfY(C, inQueue);
46:    end if

rule selectEventR(Q, E, C1, M1, X, M) =
47:  if hQueue(Q) ∧ inQueue(C1) ∧ hEvent(E) ∧ from(C1) = Q ∧ to(C1) = E ∧ evSel(M1) ∧ from(M1) =
    E ∧ to(M1) = X ∧ bool(X) then
48:    delConnectionRfrmAtob(C1, Q, E);
49:    delMappingFrmAtob(M1, E, X);
50:    addMappingFrmAtob(M, E, true);
51:    addInstanceOfY(M, evSel);
52:    end if

rule deselectEvR(E, M1, M) =
53:  if hEvent(E) ∧ evSel(M1) ∧ bool(true) from(M1) = E ∧ to(M1) = true then
54:    delMappingFrmAtob(M1, E, true);
55:    addMappingFrmAtob(M, E, false);
56:    addInstanceOfY(M, evSel);
57:    end if

rule resetR(T, M1, M2, M3, X1, X2, X3, F1, F2, F3) =
58:  if hTrans(T) ∧ fire(M1) ∧ bool(X1) ∧ from(M1) = T ∧ to(M1) = X1 ∧ fireable(M2) ∧ bool(X2) ∧
    from(M2) = T ∧ to(M2) = X2 ∧ enabled(M3) ∧ bool(X3) ∧ from(M3) = T ∧ to(M3) = X3 then
59:    delMappingFrmAtob(M1, T, X1);
60:    delMappingFrmAtob(M2, T, X2);
61:    delMappingFrmAtob(M3, T, X3);
62:    addMappingFrmAtob(F1, T, false);
63:    addMappingFrmAtob(F2, T, false);
64:    addMappingFrmAtob(F3, T, false);
65:    addInstanceOfY(Fi, fire);
66:    addInstanceOfY(Fi, enabled);
67:    addInstanceOfY(Fi, enabled);
68:    end if

8.2.6 Extended Hierarchical Automata revisited

The EHA semantics presented in the previous section was tailored to be understood by humans; there-
fore, each semantic rule was relatively simple. However, we demonstrate in Sec. 6.5.3 that such refined
rules have negative effect on verification since they produce more intermediate states and interleaving
traces. In fact, this is basically caused by the fact that the set of transitions to be fired parallelly at
a time is calculated each time from scratch although they could be derived in the initialization phase prior to execution.

For this reason, we now revisit the EHA semantics by providing a less redundant set of rules. More specifically, we introduce the concepts of configurations (states that can be active at a time) and steps (transitions that can be fired parallelly after resolving conflicts). These new concepts are summarized by the metamodel extension of EHA in Fig. 8.12.

![Diagram](image_url)

**Fig. 8.12.** Metamodel of EHA extended with steps and configurations

- An EHA is now additionally composed of steps $hStep$ and configurations $hConf$, moreover, the initial configuration $initConf$ is also identified.
- A configuration $hConf$ consists of $hStates$ (that can be active at a time). When collecting such configurations, the state hierarchy has to be traversed in a bottom-up way by extending more elementary configurations at each state:
  - For all $hStates$ which are not refined into an automaton, a new (partial) configuration is created consisting of the state as the single element.
  - When processing a $hState$ with a refined automaton, we first create the Cartesian product of configurations generated by its substates, and then we add the state itself.
  - When the top $hState$ of the hierarchy is reached, the algorithm terminates yielding all valid configurations as result.

A formalization of such an algorithm is presented in [178] by graph transformation rules.

- A step $hStep$, which is composed of non-conflicting transitions $hTrans$ and triggered by a $hEvent$, is leading from a configuration ($fromConf$) and to a configuration ($toConf$). All steps of an EHA can be generated basically by a slightly modified version of $fireFirstR$ and $fireNextR$ rules already discussed in Fig. 8.11, all other elements (like $fromConf$, $toConf$, trigger) can be derived in a straightforward way.

Now the revised operational semantics of EHA can be captured by the model transition system depicted in Fig. 8.13. Note that these rules capture the behavior of the entire system of EHA objects while the previous set of rules in Fig. 8.11 focused on the behavior of a single EHA object.

1. **enableEhaR**: A $hStep$ $T1$ is enabled if all states $S$ of its from configuration $FR$ are active, its trigger $hEvent$ $E$ is in the event queue $Q$ of the EHA $A1$, and no states LN accessed by the $islh$ guard $G$ of a $hTransition$ $T$ of the $hStep$ $T1$ are inactive (note that this condition still holds for transitions without $islh$ guards). This steps combines the effects of the previously distinct rules $selectEventR$, $enabledR$, and $fireFirstR$ as a single $hStep$ $T1$ is selected non-deterministically. If no such steps are found then the entire system is in a deadlock thus execution terminates.
2. **exitEhaR**: All the states $S$ of the $fromConf$ configuration of the firing $hStep$ $T1$ are exited by falsifying their $isAct$ attribute.
3. **enterEhaR**: All the states $S$ of the $toConf$ configuration of the firing $hStep$ $T1$ are entered by setting their $isAct$ attribute to true.
Fig. 8.13. Revised operational semantics of EHA
4. \textit{addQueueEhaR}: As the effect of firing a hStep \( T_1 \), all the the hEvents \( E \) associated to the (send) hActions \( AC \) are added to the corresponding hQueues \( Q \). This time, then the event queue is modeled as a bag and not a set by removing the previous negative condition.

5. \textit{resetEhaR}: All the dynamic attributes of a hStep \( T_1 \) are set to false, and a new step of the EHA commences.

Later, this operational semantics of EHA will serve as the basis for demonstrating different techniques concerning verification of a modeling language (in Chapter 6) and reasoning about semantics of model transformations (in Chapter 7).

\textbf{Definition 8.10 (ASM representation of EHA semantics (revisited))}. The ASM formalization of the previous semantics (Fig. 8.13) is presented below. We use the the model-level representation of VPM models this time as well for the sake of simplicity.

\begin{verbatim}
rule fireEhaRef =
  1: if try(enableEhaR) then
  2: forall(exitEhaR);
  3: forall(enterEhaR);
  4: forall(addQueueEhaR);
  5: forall(resetEhaR);
  6: end if

rule enabledR(Aa1, T1, Q, E, C_{AT}, C_{AQ}, C_{QE}, C_{TE}, M_T, F_T) =
  7: if eHA(A1) \land hStep(T1) \land steps(C_{AT}) \land from(C_{AT}) = A1 \land to(C_{AT}) = T1 \land fire(M_T) \land from(M_T) = T \land to(M_T) = false \land hQueue(Q)eventQ(C_{AQ}) \land from(C_{AQ}) = A1 \land to(C_{AQ}) = Q \land hEvent(E) \land inQueue(C_{QE}) \land from(C_{QE}) = E \land trigger(C_{TE}) \land from(C_{TE}) = T1 \land to(C_{TE}) = T \land \not\exists T, G, IN, C_{TT}, C_{TT} : (hTrans(T) \land trans(C_{TT}) \land from(C_{TT}) = T1 \land to(C_{TT}) = T \land state(IN) \land isln(C_T) \land from(C_{TT}) = T \land to(C_{TT}) = IN \land isAct(M_T) \land from(M_T) = IN \land to(M_T) = false) \land
      \not\exists FR, S, C_{TF}, C_{FS}, M_S : (hConf(FR) \land trans(C_{TF}) \land from(C_{TF}) = T1 \land to(C_{TF}) = FR \land state(S) \land
      states(C_{FS}) \land from(C_{FS}) = FR \land to(C_{FS}) = S \land isAct(M_S) \land from(M_S) = S \land to(M_S) = false)
    then
      8: delMappingFromAtoB(M_T, T1, false);
      9: addMappingFromAtoB(F_T, T1, true);
      10: addXInstanceOfY(F_T, fire);
  11: end if

rule exitEhaR(Aa1, T1, FR, S, C_{AT}, C_{AF}, C_{TF}, C_{FS}, M_T, M_S, M) =
  12: if eHA(A1) \land hStep(T1) \land steps(C_{AT}) \land from(C_{AT}) = A1 \land to(C_{AT}) = T1 \land hConf(FR) \land configs(C_{AF}) \land
      from(C_{AF}) = A1 \land to(C_{AF}) = FR \land toConf(C_{TF}) \land from(C_{TF}) = T1 \land to(C_{TF}) = FR \land state(S) \land
      states(C_{FS}) \land from(C_{FS}) = FR \land to(C_{FS}) = S \land bool(true) \land bool(false) \land fire(M_T) \land from(M_T) = T1 \land to(M_T) = true \land
      isAct(M_S) \land from(M_S) = S \land to(M_S) = true
    then
      13: delMappingFromAtoB(M_S, true);
      14: addMappingFromAtoB(M_S, false);
      15: addXInstanceOfY(M, isAct);
    end if

rule enterEhaR(Aa1, T1, TO, S, C_{AT}, C_{AF}, C_{TF}, C_{FS}, M_T, M_S, M) =
  17: if eHA(A1) \land hStep(T1) \land steps(C_{AT}) \land from(C_{AT}) = A1 \land to(C_{AT}) = T1 \land hConf(TO) \land configs(C_{AF}) \land
      from(C_{AF}) = A1 \land to(C_{AF}) = TO \land toConf(C_{TF}) \land from(C_{TF}) = T1 \land to(C_{TF}) = TO \land state(S) \land
      states(C_{FS}) \land from(C_{FS}) = TO \land to(C_{FS}) = S \land bool(true) \land bool(false) \land fire(M_T) \land from(M_T) = T1 \land to(M_T) = true \land
      isAct(M_S) \land from(M_S) = S \land to(M_S) = false
    then
      18: delMappingFromAtoB(M_S, false);
      19: addMappingFromAtoB(M, S, true);
  20: end if
\end{verbatim}
8.2 Extended Hierarchical Automata: Formal Semantics of UML Statecharts 203

20: addXInstanceOfY(\(M, is\text{Act}\));
21: end if

rule addQueEhaR(A1, T1, TR, AC, Q, E, C_{AT}, C_{TT}, C_{TA}, C_{AQ}, C_{AE}, C) =
22: if eHA(A1) ∧ hStep(T1) ∧ steps(C_{AT}) ∧ from(C_{AT}) = A1 ∧ to(C_{AT}) = T1 ∧ hTrans(TR) ∧ trans(C_{TT}) ∧
from(C_{TT}) = T1 ∧ to(C_{TT}) = TR ∧ hAction(A) ∧ effect(C_{TA}) ∧ from(C_{TA}) = TR ∧ to(C_{TA}) = Q ∧
hQueue(Q) ∧ receiver(C_{AQ}) ∧ from(C_{AQ}) = A ∧ to(C_{AQ}) = Q ∧ hEvent(E) ∧ event(C_{AE}) ∧ from(C_{AE}) =
A ∧ to(C_{AE}) = E ∧ \#C_{QE} = \text{inQueue}(C_{QE}) ∧ from(C_{QE}) = Q ∧ to(C_{QE}) = E then
23: addConnectionRfromAtomB(C, Q, E);
24: addXInstanceOfY(C, \text{inQueue});
25: end if

rule resetR(A1, T1, C, M1, X1, F1) =
26: if eHA(A1) ∧ hStep(T1) ∧ steps(C) ∧ from(C) = A1 ∧ to(C) = T1 ∧ fire(M1) ∧ bool(X1) ∧ from(M1) =
T ∧ to(M1) = X1 then
27: delMappingRfromAtomB(M1, T, X1);
28: addMappingRfromAtomB(F1, T, false);
29: addXInstanceOfY(F1, \text{fire});
30: end if

8.2.7 Correctness of EHA semantics

Now we prove that the EHA semantics of Fig. 8.11 (abbreviated as EHA^{ref}, or simply as \text{ref}) is a refinement of the EHA semantics of Fig. 8.13 (denoted as EHA^{abs}, or simply \text{abs}). Moreover, we also show that our semantics satisfies the requirements informally defined in [125] and establish a theorem conceptually similar to Theorem 1 in [105].

Equivalence of different versions of EHA semantics

In order to prove the refinement relation between the EHA^{ref} and EHA^{abs}, we assume that the steps
and configurations of EHA^{abs} (and other static derived constructs) are generated correctly. Therefore, we only focus on the equivalence of the dynamic behavior.

Since EHA^{abs} describes the behavior of a system of EHAs executed in parallel (with interleaving steps) while EHA^{ref} specifies a step in a single EHA, we further restrict our EHA^{abs} models to contain a single EHA object.

Proposition 8.11. The EHA semantics EHA^{ref} is a correct formalization of the EHA semantics EHA^{abs} in the sense that if we start from the same configuration and environment (i.e., the same set of states are active, and the same events are in the event queue) then there exists an execution path in each machine where the same states become active after executing the corresponding ASM step and the same environment is obtained. Formally, \(\forall X^{abs}, X^{ref}, \text{ref}^{abs}, \text{ref}^{ref} : X^{abs} = \text{next}_{fireEhaAbs}(X^{abs}) ∧ X^{ref} = \text{next}_{fireEhaRef}(X^{ref})\):

1. If \(\forall s, m, x : [\text{hState}(s) ∧ isAct(m) ∧ from(m) = s ∧ to(m) = x]^{ref} \iff [\text{hState}(s) ∧ isAct(m) ∧ from(m) = s ∧ to(m) = X]^{abs}\) then \(\forall s, m, x : [\forall m : \text{hState}(s) ∧ isAct(m) ∧ from(m) = s → to(m) = X]^{abs} \iff [\text{hState}(s) ∧ isAct(m) ∧ from(m) = s ∧ to(m) = X]^{refs}\).
2. If \(\forall q, e, c : [\text{hQueue}(q) ∧ hEvent(e) ∧ inQueue(c) ∧ from(c) = q ∧ to(c) = e]^{ref} \iff [\text{hQueue}(q) ∧ hEvent(e) ∧ inQueue(c) ∧ from(c) = q ∧ to(c) = e]^{abs}\) then \(\forall q, e, c : [\text{hQueue}(q) ∧ hEvent(e) ∧ inQueue(c) ∧ from(c) = q ∧ to(c) = e]^{abs} \iff [\text{hQueue}(q) ∧ hEvent(e) ∧ inQueue(c) ∧ from(c) = q ∧ to(c) = e]^{refs}\).

where \text{fireEhaAbs} \text{fireEhaRef} are the ASM rules imposed by the corresponding control structures (as defined in Def. 8.10 and Def. 8.9, respectively).
The correctness of collecting steps in EHA$^{ref}$ is captured by the following theorem, which is equivalent with Theorem 1 in [105].

**Proposition 8.12.** For all firing set $T_{fr}$ in EHA$^{ref}$ (obtained in $\mathfrak{X}_{S}^{ref}$) $T_{fr}$ is a maximal set under set inclusion, which satisfies all the following properties.

1. All transitions of $T_{fr}$ are enabled, i.e., $\forall t \in T_{fr}$: $[\varphi_{enab}(t) \equiv \text{hTrans}(T) \land \forall S : \\text{hState}(S) \land \text{srcRest}(C_{TS}) \land \text{from}(C_{TS}) = T \land \text{to}(C_{TS}) = S \land \text{isAct}(M_{S}) \land \text{from}(M_{S}) = \text{true} \land \exists E, M_{S}, C_{TS}, C_{TE} : \text{hEvent}(E) \land \text{evSel}(M_{E}) = E \land \text{to}(M_{S}) = \text{true} \land \text{trigger}(C_{TE}) \land \text{from}(C_{TE}) = T \land \text{to}(C_{TE}) = E \land \exists S_{t}, C : \text{hState}(S_{t}) \land \text{isAct}(M) \land \text{from}(M) = S_{t} \land \text{to}(M) = \text{false} \land \text{isln}(C) \land \text{from}(C) = T \land \text{to}(C) = S_{t}]^{\mathfrak{X}_{S}^{ref}}$.  

2. There is no transition outside $T_{fr}$ which is enabled in the current configuration and which has higher priority than any transition in $T_{fr}$. Formally, $\forall t' \notin T_{fr} \land [\varphi_{enab}(t') \land \exists \delta : t \in T_{fr} \land \text{givePrior}(\delta) \land \text{from}(\delta) = t \land \text{to}(\delta) = t']^{\mathfrak{X}_{S}^{ref}}$.  

3. All transitions in $T_{fr}$ respect priorities, i.e., $\forall t \in T_{fr} \forall t' \in T_{fr} : [\exists \delta : \text{givePrior}(\delta) \land \text{from}(\delta) = t \land \text{to}(\delta) = t']^{\mathfrak{X}_{S}^{ref}}$.  

4. $T_{fr}$ is conflict free, i.e., $\forall t, t' \in T_{fr} : [\exists \delta : \text{conflict}(\delta) \land \text{from}(\delta) = t \land \text{to}(\delta) = t']^{\mathfrak{X}_{S}^{ref}}$. 

### 8.3 Case Study: Verifying a Model Transformation From Extended Hierarchical Automata to Petri Nets

In this section, our aim is to transform a Petri net equivalent out of EHA models (referred to as EHA2PN model transformation) aiming to provide access to Petri net based analysis tools for UML statecharts. Prior to the detailed discussion of the transformation rules, we briefly sketch the basic modeling decisions on representing UML statecharts (more precisely, Extended Hierarchical Automata) as Petri net models.

#### 8.3.1 Modeling statecharts as Petri nets

Each EHA state is modeled with a respective place in the target Petri net model. A token in such a place denotes that the place is active, therefore, a single token is allowed on each level of the state hierarchy (forming token ring, or more formally, a place invariant). In addition, places will be generated to model messages stored in event queues of an EHA. In this sense, the presence of a message is denoted by a token in the event queue. Note that an event queue is still modeled as a set, therefore we do not need to implement FIFO handling in Petri nets.

However, if only a single place is introduced for each EHA, we lose an important piece of information, namely, the type of the message sent. Therefore, for a more faithful Petri net representation, a separate place is generated for all event types understood by an event queue, and messages therefore are dequeued from and sent to the appropriate subqueues of an state machine.

Each EHA step (which is a collection of statechart transitions that can be fired parallelly) is projected into a Petri net transition. When such a transition is fired, (i) tokens are removed from source places (i.e., places generated for the source states of the step) and event queue places, and (ii) new tokens are generated for all the target places and receiver message queues. Therefore, input and output arcs of the transition should be generated in correspondence with this rule.

**Example 8.13.** An extract of the Petri net equivalent of the voter is depicted in Fig. 8.14. For improving legibility, only two transitions (equivalents of $t_{2}$ and $t_{4}$ in the EHA model of Fig. 8.6) are shown.

- The places of the voter subsystem are constituted of the states of the voter (wait for vote, may accept, decline) and message stores for valid events (yes and no). The initial state is marked by a token in wait for vote.
8.3 Case Study: Verifying the EHA2PN Model Transformation

![Petri net model of the voter (extract)](image)

- Transition $t_2$ has two incoming arcs, one from its source state `wait_for_vote` and one from the message queue of the triggering `no_queue` event. The single outgoing place of the transition is `decline`.
- Transition $t_4$ has two incoming arcs as well, one from its source state `may_accept` and one from the message queue of the triggering `yes_queue` event. However, this transition has multiple output places: one for the target state `wait_for_vote`, and one for each target event queue that receives the generated `accept` message.

8.3.2 The EHA2PN model transformation

Prior to the transformation itself, we present a simple reference metamodel (see Fig. 8.15) that describes the interconnections allowed between EHA and Petri net elements. In contrast to the SC2EHA reference metamodel (in [178]), source and target links are more restrictive this time providing additional means to well-formedness of model transformation rules. However, this EHA2PN metamodel can also be enriched to provide more refined relations between constructs of the two languages.

![Reference metamodel of the EHA2PN transformation](image)

- A reference node of type `RefState` (which is not identical to `RefState` in the SC2EHA transformation) relates a source `hState` to a target Petri net `Place`.
- A reference node of type `RefStep` assigns a source `hStep` to a target Petri net transition (`Trans`).
- A reference node of type `RefQEvent` connects a source `hQueue` and `hEvent` to a target Petri net `Place`.

The EHA-to-Petri net transformation itself can be captured by 12 model transformation rules (shown in Fig. 8.16 and 8.17). The major conceptual difference compared with the SC2EHA model transformation is the fact that not only the static structure of models are transformed but also
dynamic constructs are handled by these rules. For the present, it can be simply interpreted as the initialization of the target Petri net model (adding tokens to the proper places). However, such dynamic interrelations form the basis of semantic projections of modeling languages later in Chapter 7.

![Diagram](image_url)

**Fig. 8.16.** From EHA to Petri nets

1. `state2placeR`: This rule expresses the fact that a Petri net place \( P \) is generated for each EHA hState \( S \) (interpreted by a reference node \( R \) of type \( \text{RefState} \)).
8.3 Case Study: Verifying the EHA2PN Model Transformation

Fig. 8.17. From EHA to Petri nets (continued)
2. **actst2tokenR:** For all EHA hStates $S$ that are active at a time (i.e., the initial states of a hAutomaton in the beginning) a token should be generated to the Petri net place $P$.

3. **passst2tokenR:** It is the negative counterpart of the previous rule stating that if a state is passive then the corresponding place $P$ contains no tokens.

4. **validate2placeR:** Each hQueue $Q$ - valid hEvent $E$ pairs are transformed into a Petri net place $P$ (interrelated by a reference node $R$ of type RefEvent).

5. **inque2tokenR:** As for the initialization (and dynamic behavior) of places related to queues, each hEvent $E$ that validly appears in a hQueue $Q$, one token is assigned to $P$.

6. **notinque2tokenR:** The negative counterpart of inque2tokenR requiring that if a certain hEvent $E$ is not in a certain queue, then the corresponding place $P$ should contain no tokens.

7. **step2transR:** For each EHA hStep ST, a Petri net transition $T$ is generated (connected via a reference node of type RefStep).

8. **fire2fireR:** If an EHA hStep ST is firing then the corresponding Petri net transition $T$ should be firing as well.

9. **stepfrom2arcR:** For each (already transformed) state $S$ that is member of the from configuration $F$ of a hStep ST (which also have a Petri net transition as image), an InArc IN is generated that leads from the corresponding Petri net place $P$ to transition $T$.

10. **stepsto2arcR:** For each state $S$ that is member of the to configuration $F$ of a hStep ST, an OutArc O is introduced that leads from the Petri net transition $T$ to place $P$.

11. **trig2arcR:** If a trigger hEvent $E$ of a hStep ST is valid in the context of a hQueue $Q$ of an EHA construct $H$ then the Petri net transition $T$ (related to hStep ST) should have an additional incoming arc $IN$ leading from the Petri net place $P$ (which is the target equivalent of the pair $(Q,E)$).

12. **eff2outarcR:** When a hEvent $E$ is sent to a target receiver hQueue $Q$ as an effect of (some transitions $T$ in) hStep ST, then the Petri net transition $T$ (related to hStep ST) should have an additional outgoing arc $O$ leading from the Petri net place $P$ (which is the target equivalent of the pair $(Q,E)$).

13. **ishin2outR:** Finally, if a hTransition TR of a hStep ST has an isIn guard $G$ that requires the presence of a hState $S$, then both an InArc $IN$ and an OutArc $O$ should be introduced between the corresponding notions (place $P$ and transition $T$) in the Petri net. In this sense, the presence of a token at the guarded place is only checked (since removed and added as well when firing the same transition).

As a result, we are able to transform a meaningful fragment of UML statecharts into Petri net models and carry out a formal analysis [133] afterwards by Petri net tools. In fact, this transformation has already been used in a Hungarian research project (IKTA 065/2000 [91]) with UML models (statecharts of a radio protocol and the monitoring system of an artificial kidney) taken from the industrial partners.

### 8.3.3 Verifying the EHA2PN transformation

In the previous sections, we have proved that the Petri net and EHA semantics is correct with respect to our informal requirements. In order to demonstrate the practical feasibility of our automated verification approach, we verify the formal correctness of the EHA2PN model transformation when transforming the simple source model of Fig. 8.5 into its Petri net equivalent (shown in Fig. 8.14) with respect to preserving the following semantic criterion.

**Definition 8.14 (Safety criterion for statecharts).** For all non-concurrent composite states in a UML statechart, only a single substate is allowed to be active at any time during execution.

**Formalizing the correctness property**

This informal requirement can be formalized by the following graphical invariant (depicted in Fig. 8.18) in the domain of Extended Hierarchical Automata (its textual equivalent logic formula is also depicted
Informally speaking, it prohibits the simultaneous activeness of two distinct substates $S1$ and $S2$ of the same $h$Automaton $A$.

$$A : hAut, S1 : hState, S2 : hState : states(A, S1) \land states(A, S2) \land isAct(S1) \land isAct(S2) \land S1 \neq S2$$

Fig. 8.18. A sample graphical safety criterion

Unfortunately, it is not at all straightforward to establish the same criterion on the meta level in the target language of Petri nets since $h$Automata are not handled by the EHA2PN transformation. However, in order to model check a certain system, a correctness criterion can be introduced on the model level. Therefore, we first automatically instantiate (the static parts of) the criterion on the concrete EHA model (as done in the transformation of graph transformation rules to transitions in Chapter 6) to obtain the model level criterion of Fig. 8.19. Note that the different (model level) patterns denote conjunctions, therefore, none of the depicted situations are allowed to occur.

$$\neg (states(a1, a1\_wait) \land states(a1, a1\_process) \land isAct(a1\_wait) \land isAct(a1\_process)) \land ...$$

Fig. 8.19. Model level safety criterion

This model level criterion is appropriate to be transformed into an equivalent criterion for the Petri net model. As $h$Automata are not projected into Petri nets, the corresponding property (shown in Fig. 8.20) contain only places having a token.

$$\neg (token(a1\_wait) = 1 \land token(a1\_process) = 1) \land ...$$

Fig. 8.20. The Petri net equivalent of the model level safety criterion
Model checking the source model

In order to verify that the safety property holds in the source EHA model, we have to generate the equivalent transition system by running the transformation of Sec. 6.3. Starting from the EHA model of Fig. 8.5 and the operational semantics of EHA (shown in Fig. 8.13), we obtain the following SAL specification as the result. Note that identifiers (i.e., values of the different domains) directly composed of the names used in Fig. 8.5. For instance, \( a_0.\\text{wait} \) denotes the hState \( wait \) of hAutomaton \( a_0 \).

\% Type declarations
hStateID : TYPE = \{a_0.\\text{wait}, a_0.\\text{mayaccept}, a_0.\\text{decline},
\hspace{1cm} a_1.\\text{wait}, a_1.\\text{process}, \ldots \};
hStepID : TYPE = \{t_1, t_2, \ldots, t_{14} \};
hEventID : TYPE = \{yes, no, decline, accept, finished \};
hQueueID : TYPE = \{v_0.q, c_1.q, c_2.q \};
ruleID : TYPE = \{enableEhaR, \ldots, resetEhaR \};
eha1 : MODULE =
BEGIN \% declaring state variables
GLOBAL inQueue: ARRAY hQueueID OF ARRAY hEventID OF BOOLEAN
GLOBAL isAct: ARRAY hStateID OF BOOLEAN
GLOBAL fire: ARRAY hStepID OF BOOLEAN
INITIALIZATION
inQueue[c_1.q][finished] = TRUE; inQueue[c_2.q][finished] = TRUE;
inQueue[v_0.q][yes] = FALSE; \ldots
isAct[a_0.\\text{wait}] = TRUE; isAct[a_0.\\text{decline}] = FALSE; \ldots
fire[t_1] = FALSE; fire[t_2] = FALSE; \ldots
TRANSACTION
\% generated for one potential matching of rule enableEhaR
pc = enableEhaR AND
fire[t_1] = FALSE AND inQueue[v_0.q][yes] AND
\hspace{1cm} NOT (isAct[a_0.\\text{wait}] = FALSE) \rightarrow
\hspace{1cm} fire'[t_1] = TRUE;
\hspace{1cm} pc' = exitEhaR;
\{ \ldots
\}
\hspace{1cm} END;

The corresponding logical formula can directly be derived from the graphical notation of the model level correctness property as follows.

eha1 \land \neg (isAct[a_0.\\text{wait}] \land isAct[a_0.\\text{decline}]) \land
\hspace{1cm} \neg (isAct[a_0.\\text{wait}] \land isAct[a_0.\\text{mayaccept}]) \land
\hspace{1cm} \neg (isAct[a_0.\\text{decline}] \land isAct[a_0.\\text{mayaccept}]) \land \ldots

A model checker easily verifies that this property holds for the EHA model.

Model checking the target model

The transition system capturing the dynamic behavior of the target Petri net model is also generated automatically (from graph transformation rules of Fig. 8.2 and the Petri net model itself of Fig. 8.14) after transforming the source EHA model into its Petri net equivalent. For better understanding, identifiers of Petri net objects are named identically to the their source correspondents. Note that place identifiers are composed of the identifiers of hStates (like \( a_0.\\text{wait} \)) and the identifiers of valid hEvent-hQueue pairs (such as \( v_{lq.\\text{yes}} \)).

\% Type declarations
placeID : TYPE = \{a_0.\\text{wait}, a_0.\\text{mayaccept}, a_0.\\text{decline}, \ldots
\hspace{1cm} v_{lq.\\text{yes}}, v_{lq.\\text{no}}, c_{lq.\\text{finished}}, \ldots \};
transID : TYPE = \{t_1, t_2, \ldots, t_{14} \};
ruleID : TYPE = \{enableTransR, delTokenR, addTokenR, delFireR \};
pr1 : MODULE =
BEGIN % declaring state variables
GLOBAL token: ARRAY placeID OF INT
GLOBAL fire: ARRAY transID OF BOOLEAN

INITIALIZATION
  token[a0_wait] = 1; token[a0_decline] = 0; ...
  token[a1_wait] = 1; token[a1_process] = 0; ...
  token[ciq_fished] = 1; token[ciq_yes] = 0; ...
  fire[t1] = FALSE; fire[t2] = FALSE; ...

TRANSITION
% generated for one potential matching of rule enableTransR
  pc = enableTransR AND
  NOT (token[a0_wait] = 0) AND
  NOT (token[ciq_yes] = 0) -->
    fire'[t1] = TRUE;
    pc' = delTokenR;
  ...
END;

% Property to be proved
pm1 |- NOT (token[a0_wait] = 1 AND token[a0_decline] = 1) AND
    NOT (token[a0_wait] = 1 AND token[a0_mayaccept] = 1) AND
    NOT (token[a0_decline] = 1 AND isAct[a0_mayaccept] = 1) AND
    ...

At this point, we need to validate whether the equality (= 1) or inequality checks (≥ 1) are required in the property to be proved. We may conclude that checking equality is probably also sufficient, however, checking inequality definitely strengthens the property, therefore we decide to prove something stronger in the Petri net model.

pm1 |- NOT (token[a0_wait] = 1 AND token[a0_decline] = 1) AND
    NOT (token[a0_wait] = 1 AND token[a0_mayaccept] = 1) AND
    NOT (token[a0_decline] = 1 AND isAct[a0_mayaccept] = 1) AND
    ...

Again, as the places derived from the states of the same hAutomaton form a place invariant (with a single token circulating around), the model checker easily verifies that strengthened property as well.

As a conclusion for this section, we may draw that our EHA2PN model transformation preserves our correctness property for the specific source EHA model and its target Petri net equivalent. Additional correctness properties can be handled similarly.

8.4 Conclusions

As a conclusion for the current chapter, I demonstrated the feasibility of the theoretical foundations of the model transformation approach (presented in Chapter 4) on industrial strength benchmarks. More specifically, I formalized well-known visual modeling languages (Petri nets and Extended Hierarchical Automata), and specified and I formally verified a complex model transformation from Extended Hierarchical Automata to Petri net models aiming to carry out Petri net based analysis of UML designs.

- Operational semantics of Petri nets. I proposed an operational semantics to the widely used visual modelling language of Petri nets [181,186] and proved that this semantics is correct with respect to the traditional semantic definition of Petri nets.
- Operational semantics of Extended Hierarchical Automata. I provided a visual semantics [168] for Extended Hierarchical Automata, a frequently used semantic domain of UML statecharts. The correctness of my approach was validated against the original proposal of Extended Hierarchical Automata in [105].
• Model transformation from EHA to Petri nets. I transformed EHA models into an equivalent Petri Net notation in order to provide access to various Petri net based formal analysis tools [91].
• Formal verification of the EHA to Petri net transformation. I formally verified the preservation of a semantic consistency property.

Practical relevance
A first practical relevance of the chapter is that we can automatically build simulatons for modeling languages using a general model transformation tool (VIATRA, for instance). We first construct an appropriate metamodel and set of graph transformation rules, and then by using the automated program generation facilities of Chapter 5, a simulator can be derived automatically. In this view, the features of existing off-the-shelf Meta-CASE tools (such as [110], for instance) can be extended to handle the dynamic behavior as well (in addition to the current support for the static structure when developing a new modeling language.

Moreover, since the semantics of these languages is more refined than the traditional formalizations (e.g., firing a transition takes several micro graph transformation steps), it can be very useful for engineers who have to learn the dynamic behavior of a new modeling paradigm (such as new diagrams in UML 2.0).

The importance of the EHA2PN model transformation is that it provides Petri net based analysis facilities following [133], for instance. In fact, this transformation has already been used in a Hungarian research project (IKTA 065/2000 [91]) with real-size UML models taken from the industrial partners (statecharts of a radio protocol and the monitoring system of an artificial kidney). This transformation also demonstrated that the run-time of a model transformation is just a few percentage of an entire verification run.

Finally, the case study on the formal verification of this model transformation demonstrated (i) how the verification framework of Sec. 7.3 is applicable on an industrial model transformation, and also (ii) how properties can be captured visually by graph patterns thus providing an alternative for textual (and sometimes very complicated) OCL expressions.

Out of scope: Component-based development

Nowadays, the component-based development of IT systems is a leading trend in software engineering. Component-based systems are rarely built from scratch, instead they are composed by the interconnection of off-the-shelf (black-box) and self-written (white-box) components. Since the new UML 2.0 has introduced the concepts of components, connectors and ports, the UML modeling framework is directly applicable to specify and design components. In this respect, our model-transformation based analysis framework can directly be used to validate design decisions in self-written components.

The plug-in of the off-the-shelf components to the VIATRA framework is also rather a modeling issue. Although the precise specification of the internal behavior (e.g., the statechart) of such a black-box component is typically unknown, the designers can easily specify the so-called protocol statecharts, which describe the process how other components should interact with the specific component. Formal analysis can be carried out following the assume-guarantee principle, i.e., we prove the correctness of the entire system while assuming that the off-the-shelf components are functioning properly. Alternatively, we may explicitly prepare a fault model that anticipates faults in black-box components as well (following the principles of [134]).

Component-based software architectures are frequently not unique but they follow a specific architectural style (such as multi-tier, client-server or service-oriented applications). Metamodeling and graph transformation techniques were applied in [16,17] to model and formally analyze by the model checking techniques of Chapter 6 (i) the reachability of certain desirable system configurations and (ii) the conformance of a given application to a specific architectural style. However, a more detailed discussion of such component-based systems are out of the scope for the current thesis for space considerations.
Conclusions

As a final conclusion, I compare the results presented in the current thesis with the main objectives (of Sec. 1.2.3). Additionally, I report on how these theoretical results have been used in practical application. I also outline some future directions of basic research and applications.

9.1 Fulfillment of Objectives

Requirement 1: Description of source and target modeling languages

I introduced the VPM metamodelling framework which is built on a minimal set of core UML/MOF elements but it extends MOF in depth by (i) a simple refinement calculus providing dynamically reconfigurable inheritance and instantiation relations for all model elements (including associations and entire metamodels), and (ii) a set of elementary operations allowing a consistent manipulation of VPM models and metamodels. As a result, transformation designers may use a much more expressive (but naturally “backward compatible”) formalism than the MOF standard to define modeling languages taken from either engineering or mathematical domains.

Requirement 2: Description of model transformations

I proposed a visual but mathematically precise specification method for model transformations based upon rule and pattern based manipulation of graph based models (as provided by the paradigm of graph transformation) constrained by a simple control flow graph. In order to provide a uniform mathematical treatment, I defined formal operational semantics to this specification technique by using abstract state machines. Since graph transformation rules themselves can also be considered as special instance models, these rules can be easily captured in traditional UML CASE tools (in addition to the system model or the metamodels of modeling languages).

Requirement 3: Back-annotation of analysis results

I defined the concepts of reference models and metamodels to interrelate source and target model elements thus supporting an efficient back-annotation of the results of a formal mathematical analysis aiming at the verification and validation of a UML based system model. As a result, critical model parts in the source UML design can easily be pinpointed to systems engineers by simply navigating through the reference model of a transformation.

However, it is still an open question (and thus primary direction to further research) how to back-annotate analysis results which are dynamic in nature. For instance, a counter example retrieved by a model checker is no longer a single model but it is a sequence of models leading from the initial source UML model to an erroneous UML model. Unfortunately, the concepts of trace diagrams are missing from UML therefore, this problem is primarily caused by UML and not by our back-annotation technique.
9 Conclusions

Requirement 4: Proving semantic consistency of transformations

I proposed a model-level verification of semantic consistency properties by enabling model checking facilities for graph transformation systems. As a result, for any specific model of an arbitrary modeling languages with semantics defined by graph transformation, domain and transformation engineers can decide with automated tool support whether a given property (e.g., safety or reachability property) holds or not. After that, if we have a pair of properties (one in the source and one in the target language) which are considered to be equivalent, the same technique can be used to verify whether a model transformation (deriving its target equivalent from a specific source model) preserves the property.

But it is an open problem to decide whether a model transformation preserves a certain property starting from any models of the source language. Unfortunately, the modeling expressiveness of model checkers is insufficient to solve this problem, therefore the use of sophisticated (and less automated) theorem provers would be required. Note, however, that in a typical design process it is sufficient to prove (using our proposed technique) that the specific source model of the UML design is correctly transformed into its target equivalent mathematical model.

Requirement 5: Automated model generation

In order to automatically generate the target mathematical model from a UML based system model, we derive an implementation of the model transformation rules by automated program generation facilities. For this purpose, I proposed that the program generation process itself can also be specified as consecutive model transformation steps. Moreover, in order to integrate our model transformation techniques directly into the MDA framework, I designed an encoding of model transformations into standard Action Semantics expressions (which is supported by several off-the-shelf UML CASE tools).

As a practical outcome, the entire model transformation process starting from the specification to the implementation of transformations can be embedded into a UML environment.

9.2 Utilization of New Results

In order to demonstrate the practical, industrial utilization of theoretical results presented in the current thesis, we (where the term “we” also refers to several students working on their Master’s or PhD thesis) carried out the development of a set of tools supporting our model transformation approach in addition to basic research.

The VIATRA model transformation framework and its applications

Adapting the main industrial standards (including XMI, MOF and UML), I designed and implemented in Prolog the VIATRA model transformation system, which carries out model transformations (following our theoretical foundations of Chapter 4 and 5) on models given in a standard XMI format by automatically generating and executing Prolog programs derived from high-level rule descriptions specified in a UML notation.

Using the VIATRA model transformation system in a Hungarian research project (IKTA 065/2000: A framework for designing and testing dependable and safety critical systems), I designed and implemented model transformations, which were tested afterwards on UML based system models taken from industrial partners. As a conclusion, we drew that even an academic prototype tool such as VIATRA had an acceptable run-time performance even for real-size models.

Modeling benchmarks

In order to demonstrate the expressiveness and practical feasibility of our model transformation approach, I proposed a formal operational semantics for the well-known visual modeling languages of
Petri nets and Extended Hierarchical Automata (which is a standard formalization of UML state-charts) in the form of model transformation systems. I also proved that this semantics is provenly equivalent in each case with the traditional mathematical description of the language.

A design methodology for model transformations

I also elaborated a design methodology for model transformations which defines the main phases of a design cycle introducing the concepts of static, dynamic and derived model elements on the metamodel level, and dividing the process of model transformations into an initialization and an execution phase.

Model checking modeling languages

We developed a tool called CheckVML [147] (based upon the foundations of Chapter 6) which automatically derives a Promela model as the output from the metamodel, the instance model and the set of graph transformation rules supplied as the input. This Promela model serves as the input of the SPIN model checker tool [87]. After that, feeding the property to be proved (given in the form of temporal logic formulae) directly to SPIN we may decide for any specific model of an arbitrary modeling language whether the given property holds.

Ongoing development and future work

An ongoing activity aims at developing a metamodeling tool based on our VPM foundations (of Chapter 2 and 3) and integrating it into the VIATRA framework afterwards. The overall objective is to develop a new generation of the so-called Meta-CASE tools which simultaneously provides mathematical preciseness, and reusability of both static and dynamic semantic descriptions for a wide range of visual modeling languages.

Further activities aim at implementing the encoding of model transformation rules into Action Semantics expressions in an off-the-shelf UML CASE tool (with support for Action Semantics). As a result, all the theoretical foundations can be integrated into an industrial environment as well.
A

The Proofs of Theorems

B.1 Proofs of Chapter 3

Proposition 3.40. Applying rule deleteFromHierarchy of Alg. 2 (referred to as dfh) in a well-formed VPM state respects Invariants 3.35–3.37. Formally, ∀\(\mathfrak{A}, \mathfrak{B} : \mathfrak{S} = \text{nextdfh}(\mathfrak{A})\) \(\land \left[\varphi_{3.35}^\text{cmd}\right]_\mathfrak{A} \rightarrow \left[\varphi_{3.36}^\text{cmd}\right]_\mathfrak{B} \land \left[\varphi_{3.37}^\text{cmd}\right]_\mathfrak{B}\).

Proof. Since the proof is identical for all cases, we only show that it preserves the validity of supertype relations (i.e., \(\left[\varphi_{3.35}^\text{cmd}\right]_\mathfrak{B}\)).

1. Reflexivity. Due to the fact that no locations of type \((X, X)\) are allowed to be modified, reflexivity trivially holds in state \(\mathfrak{B}\).

2. Transitivity. Let us suppose by contradiction that there exists \(x_1, y_1, z_1\) such that \(\text{supertype}(x_1, y_1) \land \text{supertype}(y_1, z_1) \land \neg\text{supertype}(x_1, z_1)\) in state \(\mathfrak{B}\). Since state \(\mathfrak{A}\) was consistent, rule deleteFromHierarchy was called with a parameter \(P\) (i.e., element \(P\) is being removed from the hierarchy) which is equal to either \(x_1, y_1\) or \(z_1\). However, if \(P = x_1 \lor P = y_1\) then relation \(\text{supertype}(x_1, y_1)\) should have been falsified and if \(P = z_1\) then location \(\text{supertype}(y_1, z_1)\) should have been set to false, which contradicts our assumption.

3. Anti-symmetry. Since state \(\mathfrak{A}\) is consistent and no new locations of supertype are set to true therefore antisymmetry is trivially preserved. \(\square\)

Proposition 3.42. Supposing that deleteConnection (deleteMapping) of Alg. 12 soundly removes a connection (a mapping) from a VPM state (which we prove later in Prop. 3.65–3.66), the rule application of deleteDanglingEdgesOfX of Alg. 3 (referred to as gdde(X)) on entity X in a well-formed VPM state respects Invariant 3.32. Formally, ∀\(\mathfrak{A}, \mathfrak{B} : \mathfrak{S} = \text{nextgdde}(\mathfrak{A})\) \(\land \left[\varphi_{3.38}^\text{loc}\right]_\mathfrak{A} \rightarrow \left[\varphi_{3.39}^\text{loc}\right]_\mathfrak{B} \land \left[\text{isIsolated}(X)\right]_\mathfrak{B}\).

Similarly, the rule application of deleteDanglingEdgesOfXY on model elements X and Y (referred to as idde(X,Y)) in a well-formed VPM state respects Invariant 3.32. Formally, ∀\(\mathfrak{A}, \mathfrak{B} : \mathfrak{S} = \text{nextidde}(\mathfrak{A}, \mathfrak{Y})\) \(\land \left[\varphi_{3.38}^\text{loc}\right]_\mathfrak{A} \rightarrow \left[\varphi_{3.39}^\text{loc}\right]_\mathfrak{B} \land \left[\text{isXIsolated}(X,Y)\right]_\mathfrak{B}\).

Proof. Since state \(\mathfrak{A}\) was consistent, and rule deleteDanglingEdgesOfX removes edges consistently (according to our assumption), moreover, only edges are deleted but not nodes, state \(\mathfrak{B}\) is consequently consistent.

Now we should prove that applying this rule on an entity \(x_1\) results in an isolated entity (shown only for connections and the global case).

1. Let us suppose by contradiction that \(\left[\text{isIsolated}(X)\right]_\mathfrak{B} = false\), e.g., let \(r_1\) be a connection such that \(\left[\text{from}(r_1) = x_1\right]_\mathfrak{B}\).

2. However, \(r_1\) is an appropriate connection that may correspond to \(Q\) in deleteDanglingEdgesOfX, therefore, the removal of \(r_1\) is inevitably prescribed. As deleteConnection(Q) is correct by assumption, this contradicts our indirect assumption and completes our proof. \(\square\)
Proposition 3.43. Algorithm 4 always terminates.

Proof. This proposition is a trivial consequence of the fact that at least one new location is falsified at each step of the algorithm, and no locations are ever set to true. Therefore as a state 𝛃 of the VPM framework is finite in the sense that only finitely many locations of a characteristic function is allowed to be defined (i.e., 𝑓(𝑥) ≠ false), the algorithm will terminate after finitely many steps. □

Proposition 3.44. The result of applying the ASM rule checkConsistency of Alg. 4 (abbreviated as cc) always fulfills Invariant 3.3′ (regardless of the pre-state). Formally, ∀ 𝛃, 𝛃 : 𝛃 = nextₜ(𝛃) → [𝜑_{3.34}^{sup} ∧ 𝜑_{3.34}^{in} ≅ \mathbb{B}].

Proof. The proof is rather straightforward since the same subformulæ are used in Invariant 3.34 and Algorithm 4.

The implication of 𝜑_{3.34}^{sup} only becomes false if its antecedent formula supertype(𝑋, 𝑌) is true but its consequent formula is supertype(𝑋, 𝑌) is false. Let us suppose by contradiction that Algorithm 4 terminates in a state 𝛃₁ where (𝑥₁, 𝑦₁) is a pair that still falsifies the implication.

However, as model element 𝑦₁ is not a supertype of 𝑥₁, according to is supertype(𝑋, 𝑌) (i.e., the refinement axioms) but supertype(𝑋, 𝑌) holds then according to Line 1 of Alg. 4, the corresponding location supertype(𝑥₁, 𝑦₁) should have been falsified by the algorithm. This contradicts our assumption that the algorithm terminated in 𝛃₁.

Essentially, we should proceed with the proof of instantiations (instanceOf relations) in a similar way. □

Proposition 3.45. Applying rules addYtoSupertypeOFX (referred to as add) and delZfromSupertypeOFX (referred to as del) of Alg. 5 in a well-formed VPM state preserves Invariants 3.3′ and 3.35. Formally, ∀ 𝛃, 𝛃 : (𝛃 = nextₜ(𝛃) ∨ 𝛃 = nextₜ(𝛃)) ∧ [𝜑_{3.34}^{sup} ≅ \mathbb{B}] → [𝜑_{3.34}^{sup} ≅ \mathbb{B}] ∧ [𝜑_{3.35}^{po} ≅ \mathbb{B}]

Proof. Let us first investigate rule addYtoSupertypeOFX (add).

1. Since [𝜑_{3.34}^{sup} ≅ \mathbb{B}] holds and a location (𝑋, 𝑌) of supertype is only set to true if the truth of is supertype(𝑋, 𝑌) is guaranteed therefore [𝜑_{3.34}^{sup} ≅ \mathbb{B}] trivially holds in a corresponding next state 𝛃.
2. For proving [𝜑_{3.35}^{po} ≅ \mathbb{B}], we should consider the following.

a) Reflexivity. No locations (𝑋, 𝑋) are allowed to be modified in supertype relation therefore [𝜑_{3.35}^{re} ≅ \mathbb{B}] trivially holds;

b) Transitivity. If we intend to set a location 𝑋, 𝑌 to true then all locations 𝑍 that are supertypes of 𝑌 are also set to true (Line 2-3) which proves [𝜑_{3.35}^{trans} ≅ \mathbb{B}];

c) Anti-symmetry. Due to Line 1, circularities are prohibited therefore [𝜑_{3.35}^{anti} ≅ \mathbb{B}] also holds.

Now we prove the correctness of rule delZfromSupertypeOFX (del).

1. Since the falsification of certain supertype locations may implicitly falsify in turn additional supertype locations, we guarantee the validity of [𝜑_{3.34}^{sup} ≅ \mathbb{B}] by an explicit call to checkConsistency().
2. For proving [𝜑_{3.35}^{po} ≅ \mathbb{B}], we should again consider that

a) Reflexivity. No locations (𝑋, 𝑋) are modified in supertype relation therefore [𝜑_{3.35}^{re} ≅ \mathbb{B}] trivially holds;

b) Transitivity. If a location (𝑋, 𝑌) is intended to be falsified then all locations 𝑍 that are supertypes of 𝑋 but subtypes of 𝑌 are also falsified (Line 9-10) which proves [𝜑_{3.35}^{trans} ≅ \mathbb{B}];

c) Anti-symmetry. Due to Line 8, circularities are prohibited therefore [𝜑_{3.35}^{anti} ≅ \mathbb{B}] also holds.

Proposition 3.48. Applying rules addXtoComponentOFY (below referred to as add) and delXfromComponentOFZ (referred to as del) of Alg. 6 in a well-formed VPM state preserves Invariant 3.37. Formally, ∀ 𝛃, 𝛃 : (𝛃 = nextₜ(𝛃) ∨ 𝛃 = nextₜ(𝛃)) ∧ [𝜑_{3.38}^{comp} ≅ \mathbb{B}] → [𝜑_{3.37}^{comp} ≅ \mathbb{B}].
Proof. As being conceptually very similar to the proof of Proposition 3.45 (reflexivity and transitivity should be proved again) we skip the proof for space considerations. □

**Proposition 3.51.** Applying rule addEntity (referred to as ae) of Alg. 7 in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : (\mathcal{B} = \text{next}_{ae}(\mathcal{A}) \lor \mathcal{B} = \text{next}_{de}(\mathcal{A})) \land [\varphi_{3.38}^{\text{con}}]_{\mathcal{A}} \rightarrow [\varphi_{3.38}^{\text{con}}]_{\mathcal{B}} \)

Proof. This proposition is rather straightforward since only locations of type \((X, X)\) were modified in supertype, instanceOf and componentOf relations, which preserves reflexivity. Moreover, such new relations cannot contradict \(\varphi_{3.34}^{ref}\), since \(E \sqsubseteq E\) always holds in VPM (and \(\varphi_{3.4}^{ref}\) is a faithful representation of VPM axioms). Finally, we notice that these modifications do not influence the truth of any other invariants. □

**Proposition 3.53.** Applying rule softDeleteEntity of Alg. 8 (referred to as sde) in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : \mathcal{B} = \text{next}_{sde}(\mathcal{A}) \land [\varphi_{3.38}^{\text{con}}]_{\mathcal{A}} \rightarrow [\varphi_{3.38}^{\text{con}}]_{\mathcal{B}} \).

Proof. As rule deleteFromHierarchy is called first, it immediately implies \([\varphi_{3.38}^{\text{con}}]_{\mathcal{B}}\), \([\varphi_{3.38}^{\text{con}}]_{\mathcal{B}}\) and \([\varphi_{3.37}^{\text{comp}}]_{\mathcal{B}}\) (as proved in Prop. 3.40).

Then since checkConsistency is called finally, it implies the truth of \([\varphi_{3.34}^{ref}]_{\mathcal{B}}\) (due to Prop. 3.44).

Therefore, we only have to show that the result of the application of softDeleteEntity is a well-formed global graph (\(\varphi_{3.32}^{\text{deg}}\)). Equivalently, it is sufficient to prove that an isolated entity can safely be removed from the model space. As connections and mappings are treated identically, we only show it for the “connection” case.

1. Let us suppose by contradiction that there exists a connection \(r_1\) violating Invariant \(\varphi_{3.32}^{\text{con}}\), e.g., \(\exists X : \text{entity}(X) \land \text{from}(r_1) = X\) in state \(\mathcal{B}\).
2. Since state \(\mathcal{A}\) was consistent, there was an entity \(x_1\) for which \([\text{from}(r_1) = x_1]\). As a consequence, we should have applied rule softDeleteEntity with parameter \(x = x_1\) otherwise entity \(x_1\) had not been removed when getting to state \(\mathcal{B}\).
3. However, as a precondition of this rule application on \(x_1\), entity \(x_1\) should be isolated in state \(\mathcal{A}\), which means that \(\forall Y : (\text{connection}(Y) \lor \text{mapping}(Y)) \rightarrow (\text{from}(Y) \neq X \lor \text{to}(Y) \neq X)\).
4. As \(\text{from}(r_1) = x_1\) (i.e., \(r_1\) is a wrong \(Y\)), this is a contradiction with our indirect assumption.

Since rule softDeleteEntity does not have any impact on the truth of any other invariants this finishes our proof. □

**Proposition 3.55.** Applying rule forcedDeleteEntity of Alg. 8 (referred to as fde) in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : \mathcal{B} = \text{next}_{fde}(\mathcal{A}) \land [\varphi_{3.38}^{\text{con}}]_{\mathcal{A}} \rightarrow [\varphi_{3.38}^{\text{con}}]_{\mathcal{B}} \).

Proof. A direct consequence of Prop. 3.53 and Prop. 3.42. □

**Proposition 3.63.** Applying rule addConnectionRfromAtoB (referred to as gac) of Alg. 11 in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : \mathcal{B} = \text{next}_{add}(\mathcal{A}) \land [\varphi_{3.38}^{\text{con}}]_{\mathcal{A}} \rightarrow [\varphi_{3.38}^{\text{con}}]_{\mathcal{B}} \).

Applying rule addConnectionRfromAtoBinE (referred to as lac) of Alg. 11 in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : \mathcal{B} = \text{next}_{lac}(\mathcal{A}) \land [\varphi_{3.38}^{\text{con}}]_{\mathcal{A}} \rightarrow [\varphi_{3.38}^{\text{con}}]_{\mathcal{B}} \).

Proof. The proof of Prop. 3.51 can be repeated with a single extension, namely, proving that the result of rule application is still a well-formed graph (Invariant 3.32). However, as state \(\mathcal{A}\) is consistent and connection \(R\) is non-existent in \(\mathcal{A}\), but entities \(A\) and \(B\) do exist, the rule adds a valid edge to the global VPM graph with \(A\) as the source node and \(B\) as the target node, and \([\varphi_{3.33}^{\text{deg}}]_{\mathcal{B}}\) trivially holds.

Finally, note that the local case can be treated similarly. □
Proposition 3.65. Applying rule delConnectionRfromAtoB (referred to as gcd) of Alg. 12 in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : (\mathcal{B} = \text{next}_{\text{gcd}}(\mathcal{A}) \lor \mathcal{B} = \text{next}_{\text{del}}(\mathcal{A})) \land [\varphi_{3.38}^{\mathcal{A}}]_{\zeta}^{\mathcal{B}} \land [\text{connection}(R)]_{\zeta}^{\mathcal{B}} = \text{false} \) \\
Applying rule delConnectionRfromAtoBinE (referred to as ldc) of Alg. 12 in a well-formed VPM state preserves Invariant 3.38. Formally, \( \forall \mathcal{A}, \mathcal{B} : (\mathcal{B} = \text{next}_{\text{ldc}}(\mathcal{A}) \lor \mathcal{B} = \text{next}_{\text{del}}(\mathcal{A})) \land [\varphi_{3.38}^{\mathcal{A}}]_{\zeta}^{\mathcal{B}} \rightarrow [\varphi_{3.38}^{\mathcal{B}}]_{\zeta}^{\mathcal{B}} \).

Proof. Again we only prove the global case for the sake of simplicity. The proof itself is partially identical to the proof of Prop. 3.53. We only need to show that the VPM model of state \( \mathcal{B} \) is a well-formed graph \( (\varphi_{3.32}^{\mathcal{B}}) \), and an existing connection \( R \) is, in fact, removed as result of rule application.

Equivalently, it is sufficient to prove that a well-formed connection can safely be removed from the model space.

1. Let us suppose by contradiction that there is a connection \( r_1 \) (i.e., \([\text{connection}(r_1)]_{\zeta}^{\mathcal{B}} \) violating Invariant \( \varphi_{3.32}^{\mathcal{B}} \), e.g., \( \exists X : \text{entity}(X) \land \text{from}(r_1) = X \) in state \( \mathcal{B} \).
2. Since state \( \mathcal{A} \) was consistent, there was an entity \( x_1 \) for which \( \text{from}(r_1) = x_1 \) and \( \text{to}(r_1) = y_1 \) with existing entities \( X \) and \( Y \).
3. As a consequence, we had to apply rule delConnectionRfromAtoB with parameters \( (R = r_1, X = x_1, Y = y_1) \) that undefined location \( \text{from}(r_1) = x_1 \) (note that entity \( x_1 \) cannot be removed).
4. However, this is a contradiction with the indirect assumption that all preconditions of delConnectionRfromAtoB are satisfied, therefore Line 3 of Alg. 12 falsifies connection \( r_1 \) in state \( \mathcal{B} \), i.e., \([\neg\text{connection}(r_1)]_{\zeta}^{\mathcal{B}} \).

B.2 Proofs of Chapter 4

Proposition 4.22. Let \( r = (\text{Pre}, \text{Post}, \text{par}) \) be a graph transformation rule (with Lhs as the root element of Pre) applied in a consistent VPM model state \( \mathcal{A} \) to a matching pattern given by the variable assignment \( \zeta \) (i.e., \([\varphi_{\text{Pre,par}}]_{\zeta}^{\mathcal{A}} = \text{true} \)). Then for all \( \mathcal{B} = \text{next}_{\text{delete}}(\mathcal{A}, \zeta) \),

1. state \( \mathcal{B} \) is consistent
2. all elements \( q \) in \( \text{Lhs} \setminus \text{Post} \) that are matched in \( \mathcal{A} \) are successfully removed, i.e.,
   a) \( \forall e \in \text{Lhs} \land e \notin \text{Post} : [\text{entity}(E)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   b) \( \forall c \in \text{Lhs} \land c \notin \text{Post} : [\text{connection}(C)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   c) \( \forall m \in \text{Lhs} \land m \notin \text{Post} : [\text{mapping}(M)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   d) \( \forall a \rightarrow b \in \text{Lhs} \land a \notin \text{Post} : [\text{supertype}(A,B)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   e) \( \forall a \rightarrow b \in \text{Lhs} \land a \notin \text{Post} : [\text{instanceOf}(A,B)]_{\zeta}^{\mathcal{B}} = \text{false} \)
   f) \( \forall a = b[a] \in \text{Lhs} \land a = b[a] \notin \text{Post} : [\text{componentOf}(A,B)]_{\zeta}^{\mathcal{B}} = \text{false} \).

As an abbreviation we write \( \forall q \in \text{Lhs} \land q \notin \text{Post} : [\neg\text{element}(Q)]_{\zeta}^{\mathcal{B}} \) to denote the different cases uniformly.
3. If the dangling condition is also prescribed (i.e., \([\varphi_{\text{Pre,par}}]_{\zeta}^{\mathcal{A}} = \text{true} \)) and no forcedDeleteEntity operations are used then all graph element \( x \) (i.e., entity, connection or mapping) that was removed as a result of rule application should be explicitly deleted by \( r \). Formally,
   a) \( \forall E : [\text{entity}(E)]_{\zeta}^{\mathcal{B}} \land [\neg\text{entity}(E)]_{\zeta}^{\mathcal{B}} \rightarrow E \in \text{Del}_r \)
   b) \( \forall C : [\text{connection}(C)]_{\zeta}^{\mathcal{B}} \land [\neg\text{connection}(C)]_{\zeta}^{\mathcal{B}} \rightarrow C \in \text{Del}_r \)
   c) \( \forall M : [\text{mapping}(M)]_{\zeta}^{\mathcal{B}} \land [\neg\text{mapping}(M)]_{\zeta}^{\mathcal{B}} \rightarrow M \in \text{Del}_r \).

As an abbreviation we write \( \forall Q : [\text{element}(Q)]_{\zeta}^{\mathcal{B}} \land [\neg\text{element}(Q)]_{\zeta}^{\mathcal{B}} \rightarrow Q \in \text{Del}_r \) to denote the different cases uniformly.
Proof (Sketch). For space considerations, we only prove the theorem in the case of entities (while the rest of the unhandled cases can be treated similarly).

1. The result state $\mathcal{B}$ is consistent. Since we use the elementary VPM operations to manipulate the model that were proved to be sound in Sec. 3.4 (as a result of Propositions and Corollaries 3.40–3.66), that guarantees the consistency of state $\mathcal{B}$.

2. Everything that is prescribed by $r$ is removed. The elementary VPM operations guarantee the removal of an element $q$ provided that (i) we call them with proper parameters, and (ii) certain preconditions (i.e., antecedent formulæ) of Corollaries 3.46–3.66 hold.

   a) Forced delete. As a consequence of Table 4.3, if an entity $e \in Lhs$ but $e \notin Post$ then elementary VPM operation forcedDeleteEntity is called with parameter $E = e.id$ or $E = X_e$ with $\{X_e \mapsto e.id\} \in \zeta$, which proves that the proper parameter was passed. Moreover, due to the successful pattern matching phase, entity$(E)$ should also hold which satisfies the precondition of Corollary 3.56 and completes the proof of this case; since $\forall \mathcal{A}, \mathcal{B} : \mathcal{B} = \text{next} \downarrow \text{de}(\mathcal{A}, \zeta) \rightarrow [\text{entity}]^\mathcal{B}_\zeta = \text{false}$.

   b) SoftDelete. In case of softDeleteEntity, we need to show in addition that entity $e$ is isolated before calling the rule. However, $e$ satisfies the dangling condition $\psi_{e, f}$ and $\psi_{e, f}$. Hence

   - $e$ is either isolated prior to rule application (i.e., $\|\forall X : \text{from}(X) = E \lor \text{to}(X) = E\|^\mathcal{B}_\zeta$ where $x$ is a connection or a mapping), or
   - $e$ is not isolated (i.e., $\|\exists X : \text{from}(X) = E \lor \text{to}(X) = E\|^\mathcal{B}_\zeta$), but $x \in (Lhs \setminus Post)$ thus $X \in \text{Del}_r$. However, since the deletion of connections and mappings precedes the deletion of entities, all such connection/mapping $x$ is definitely removed before calling $\text{del}_e$ (according to the definition of rule delete).

3. Nothing but what prescribed by $r$ is removed. We only need to consider that (due to the construction of elementary VPM operations) merely rule forcedDeleteEntity (which we eliminated as an assumption) may violate our conditions as it removes all dangling edges which are not necessarily defined by rule $r$ before calling softDeleteEntity. Other rules may only remove refinement and component-of relations as side effects but they never modify two different kind of elements (e.g., entities and connections).

\[\Box\]

Proposition 4.25 (Correctness of addition). Let $r = (\text{Pre}, \text{Post}, \text{par})$ be a graph transformation rule (with $Lhs$ as the root element of $Pre$) applied in a consistent VPM model state $\mathcal{B}$ and an extended matching $\zeta$. Then for all $\mathcal{C} = \text{next} \downarrow \text{add}(\mathcal{B}, \zeta)$,

1. state $\mathcal{C}$ is consistent

2. if all graph elements $q \in \text{Post} \setminus Lhs$ (i.e., entities, connections and mappings) that are aimed to be created are non-existent then the corresponding new elements are successfully added. Formally,

   a) $\forall e \notin Lhs \land e \in \text{Post} : \|\neg\text{entity}(E)\|_{\zeta}^{\mathcal{C}} \rightarrow \|\text{entity}(E)\|_{\zeta}^{\mathcal{C}}$

   b) $\forall e \notin Lhs \land c \in \text{Post} : \|\neg\text{connection}(C)\|_{\zeta}^{\mathcal{C}} \rightarrow \|\text{connection}(C)\|_{\zeta}^{\mathcal{C}}$

   c) $\forall m \notin Lhs \land c \in \text{Post} : \|\neg\text{mapping}(M)\|_{\zeta}^{\mathcal{C}} \rightarrow \|\text{mapping}(M)\|_{\zeta}^{\mathcal{C}}$

   As an abbreviation we write $\forall q \notin Lhs \land q \in \text{Post} : \|\neg\text{element}(Q)\|_{\zeta}^{\mathcal{C}} \rightarrow \|\text{element}(Q)\|_{\zeta}^{\mathcal{C}}$

3. Nothing else but what prescribed by Post $\setminus Lhs$ is created as a result of rule application. Formally,

   a) $\forall E : \|\neg\text{entity}(E)\|_{\zeta}^{\mathcal{B}} \land \|\text{entity}(E)\|_{\zeta}^{\mathcal{C}} \rightarrow E \in \text{Add}_{r}$

   b) $\forall C : \|\neg\text{connection}(C)\|_{\zeta}^{\mathcal{B}} \land \|\text{connection}(C)\|_{\zeta}^{\mathcal{C}} \rightarrow C \in \text{Add}_{r}$

   c) $\forall M : \|\neg\text{mapping}(M)\|_{\zeta}^{\mathcal{B}} \land \|\text{mapping}(M)\|_{\zeta}^{\mathcal{C}} \rightarrow M \in \text{Add}_{r}$

   As an abbreviation we write $\forall Q : \|\neg\text{element}(Q)\|_{\zeta}^{\mathcal{B}} \land \|\text{element}(Q)\|_{\zeta}^{\mathcal{C}} \rightarrow Q \in \text{Add}_{r}$.

Proof (Sketch). For space considerations, we only prove the theorem for the case of entities.
1. The result state $\mathcal{C}$ is consistent. A direct consequence of using consistency preserving elementary VPM operations to manipulate the model.

2. Everything that is prescribed by $r$ is added. The elementary VPM operations guarantee the proper addition of an element $q$ provided that (i) we call them with valid parameters, and (ii) certain preconditions (i.e., antecedent formulae) of Corollaries 3.46–3.66 hold.

As a consequence of Table 4.4, if an entity $e \in \text{Post}$ but $e \notin \text{Lhs}$ then the elementary VPM operation $\text{addEntity}$ is called with a parameter that is either a constant $E = e.id$, or a variable $E = X_e$ with $\{X_e \mapsto e.id\} \in \zeta$, which proves that a valid parameter is passed. Moreover, the preconditions of $\text{addEntity}$ is explicitly assumed, which completes the proof of this case since $\forall \mathfrak{A}, \mathfrak{B} : \mathfrak{B} = \text{next}_{\text{add}}(\mathfrak{A}, \zeta) \land \langle \neg \text{element}(E) \rangle^\mathfrak{A}_\zeta \rightarrow \langle \text{element}(E) \rangle^\mathfrak{C}_\zeta$

For the case of mappings and connections, we need to consider that, since the addition of entities precedes the creation of connections and mappings, the preconditions of those operations (e.g., $\text{addConnectionRToAtoB}$) will definitely hold when they are called in turn.

3. Nothing but what prescribed by $r$ is added. This is a trivial consequence of the construction of elementary VPM operations (we may delete elements as side effects but implicit addition is never allowed).

**Theorem 4.28 (Correctness of a graph transformation step).** Let $r = (\text{Pre}, \text{Post}, \text{par})$ be a graph transformation rule (with Lhs as the root element of Pre and with only variables in Post \ Lhs), and $\mathfrak{A}$ a consistent VPM model state. Moreover, let $\zeta$ be an extended matching that contains an assignment at least for variables $\overline{X}_{\text{Lhs}}$ and $\overline{X}_{\text{add}}$.

Now if $\langle \phi_{\text{Lhs}}(\overline{X}_{\text{Lhs}}) \rangle^\mathfrak{A}_\zeta \land \varphi_r \land \delta_r \rangle^\mathfrak{C}_\zeta$ is true then for all states $\mathcal{C} = \text{next}_{\text{add}}(\mathfrak{A}, \zeta)$ the following holds.

1. $\forall q \in \text{Lhs} \land q \notin \text{Post} : \langle \neg \text{element}(Q) \rangle^\mathfrak{C}_\zeta$ (all elements mapped to a rule element in Lhs \ Post are removed)

2. $\forall Q : \langle \text{element}(Q) \rangle^\mathfrak{A}_\zeta \land \langle \neg \text{element}(Q) \rangle^\mathfrak{C}_\zeta \rightarrow Q \in \text{Del}_r$ (nothing else is removed)

3. $\forall q \notin \text{Lhs} \land q \in \text{Post} : \langle \text{element}(Q) \rangle^\mathfrak{C}_\zeta$ (all elements mapped to a rule element in Post \ Lhs are created)

4. $\forall Q : \langle \neg \text{element}(Q) \rangle^\mathfrak{A}_\zeta \land \langle \text{element}(Q) \rangle^\mathfrak{C}_\zeta \rightarrow Q \in \text{Add}_r$ (nothing else is created).

Now if $\langle \phi_{\text{Lhs}}(\overline{X}_{\text{Lhs}}) \rangle^\mathfrak{A}_\zeta$ is true then for all states $\mathcal{C} = \text{next}_{\text{add}}(\mathfrak{A}, \zeta)$ Statements 1, 3, and 4 hold.

**Proof.** Due to Propositions 4.22–4.26, all Statements 1–4 hold since $\zeta$ is an extended matching as we can apply first the results of Prop. 4.22 to obtain an intermediate state $\mathfrak{B}$ and then apply the results to this intermediate state in order to obtain the final result $\mathcal{C}$. □

**Proposition 4.30.** The variable assignment $\zeta$ defined by the **choose** and **create** constructs is an extended matching.

**Proof.** A variable assignment $\zeta$ is an extended matching (by definition) if it is a matching and it contains assignments for variables associated to freshly added elements, i.e., $\langle \phi_{\text{Lhs}}(\overline{X}_{\text{Lhs}}) \rangle^\mathfrak{A}_\zeta = \text{true} \land \forall q \in \text{Post} \land q \notin \text{Lhs} : Q = X_q \rightarrow \exists q.id : \{X_q \mapsto q.id\} \in \zeta$.

- Now we should consider that $\langle \phi_{\text{Lhs}}(\overline{X}_{\text{Lhs}}) \rangle^\mathfrak{A}_\zeta = \text{true}$ in both the SPO and DPO case therefore the **choose** construct (which assigns a constant value to each variable in Lhs) defines an assignment $\zeta$ that is a matching.

- Moreover, each element in Post \ Lhs is a variable, therefore the **create** construct extends $\zeta$ to contain fresh elements taken from the reserve of the superuniverse.

**B.3 Proofs of Chapter 5**

**Proposition 6.14 (Correctness of Prolog rules).** Let $\mathfrak{A}^\mathfrak{p}$ be the state of the Prolog database, and $\mathcal{F}(\mathfrak{A}^\mathfrak{p}) = \mathfrak{A}^\mathfrak{p}$ be its equivalent in the model level ASM representation of VPM models. Moreover, let $r^p$ be the Prolog representation of a graph transformation rule $r^d$ submitted as a query to the Prolog engine.
1. If the Prolog execution succeeds with a result state $\mathcal{V}_r$ (denoted as $l h s_r \rightarrow \mathcal{V}_r$) then there exists a state $\mathcal{A}_d$ such that $\mathcal{V}_r = \mathcal{F}(\mathcal{A}_d)$ and $\mathcal{A}_d = \text{next}_{\mathcal{V}_r}(\mathcal{A}_d)$.

2. If the Prolog execution fails, then there are no states $\mathcal{A}_d \neq \mathcal{A}_r$ such that $\mathcal{A}_d = \text{next}_{\mathcal{V}_r}(\mathcal{A}_r)$.

Proof. For the first part of the theorem, we have to prove the following cases.

1. If all variables in the LHS part of the Prolog rule (denoted as $l h s_r$) of the ASM representation of $r^d$ are successfully substituted by unifying dynamic clauses, then the same assignment constitutes a matching in the GT rule;

   • Since the initial state of the computation ($\mathcal{A}_d$ and $\mathcal{A}_r$) is equivalent, moreover, the Prolog encoding of the model exactly follows the ASM representation of VPM models, therefore if a clause $p(X)$ in the LHS of the Prolog rule is unified with dynamic clause $p(a)$ in the Prolog model $\mathcal{A}_r$, then $[p(a)]_{\mathcal{A}_d} = \text{true}$.

   • Therefore, one can select $a$ to instantiate the corresponding variable (by the choose construct) on the LHS of (the ASM representation of) $r^d$. Therefore, if the Prolog computation gets through the LHS clauses (i.e., $l h s_r$), then it implies the truth of $\phi_{l h s}$.

2. If the Prolog computation gets through the negative application condition code part $n e g_r$, then the negative conditions are respected in the GT rule as well.

   • Let us suppose by contradiction that there exists an assignment which violates the negative condition of $r^d$, but there is no such violating substitution for $r^r$.

   • Let a $p(X)$ be a term in the NEG graph $\phi_{neg}$, and $\zeta = \{X \mapsto a, \ldots\}$ is a (violating) assignment. Then $[p(a)]_{\zeta} = true$, and due to the consistency of $\mathcal{A}_d$ and $\mathcal{A}_r$, $p(X)$ can be unified with $p(a)$ in $r^r$.

   • Since this holds for all terms in $\phi_{neg}$, we have a contradiction with our indirect assumption, since $\zeta$ becomes a substitution fulfilling all clauses in $n e g_r$, in which case backtracking is initiated in the computation not get through the negative part $n e g_r$.

3. The modifications (assertions, retractions) carried out by the Prolog computation are performed through the GT rule as well (by corresponding additions and deletions).

   • This trivially holds, if the Prolog implementation is directly based upon the same VPM operations of Sec. 3.4.

For the second part of the theorem, we may use a similar reasoning.

1. If the computation fails to substitute all variables in the LHS part $l h s_r$ of the Prolog rule, then no assignments exist that satisfy the LHS of rule $r^d$.

   • Let us suppose indirectly that there exists an assignment which satisfies the LHS of $r^d$ but there is no such satisfying substitution for $r^r$.

   • Let a $p(X)$ be a term in the LHS graph $\phi_{l h s}$, and $\zeta = \{X \mapsto a, \ldots\}$ is a (satisfying) assignment. Then $[p(a)]_{\zeta} = true$, and due to the consistency of $\mathcal{A}_d$ and $\mathcal{A}_r$, $p(X)$ can be unified with $p(a)$ in $r^r$.

   • Since this holds for all terms in $\phi_{l h s}$, we have a contradiction with our indirect assumption, since $\zeta$ becomes a substitution fulfilling all clauses in $l h s_r$, in which case the computation gets through the LHS part $l h s_r$ and thus fails to substitute all variables.

2. If the computation respects the LHS of the Prolog rule $r^r$ but there is a substitution violating the negative condition, then this is a assignment violating the negative condition in $r^d$ as well.

   • Due to the consistency of $\mathcal{A}_d$ and $\mathcal{A}_r$, one can select the same values violating the NEG part of $r^r$ to instantiate the corresponding variable by the choose construct in the NEG part of $r^d$.

Proposition 6.15 (Correctness of try). If $r^r$ is a correct implementation of $r^d$ (see Prop. 5.14 above), then $\text{try}(r^r)$ (see Table 5.1) is a correct implementation of $\text{try}(r^d)$ (see Def. 4.4.2).
Formally, let \( \mathcal{A}^pr \) be the state of the Prolog database, and \( \mathcal{F}(\mathcal{A}^d) = \mathcal{A}^pr \) be its equivalent in the model-level ASM representation of VPM models, and let \( \text{try}(r^{pr}) \) be the query submitted to the Prolog engine. If the Prolog execution succeeds with a result state \( \mathcal{B}^pr \) (denoted as \( \mathcal{A}^pr \Rightarrow \mathcal{B}^pr \)) then there exists a state \( \mathcal{B}^d \) such that \( \mathcal{B}^pr = \mathcal{F}(\mathcal{B}^d) \) and \( \mathcal{B}^d = \text{next}_{\text{try}}(r^{pr})(\mathcal{A}^d) \).

**Proof.** If \( r^{pr} \) succeeds then (due to Prop. 5.14) \( r^d \) can also be applied successfully. However, in such a case, \( \text{try}(r^d) \) also succeeds, and returns with true.

Moreover, if \( r^{pr} \) succeeds then \( \text{try}(r^{pr}) \) cuts off all choicepoints inside \( r^{pr} \), therefore, the control will never again return into \( r^{pr} \) for the same matching. \( \Box \)

**Proposition 6.16 (Correctness of nondet).** If \( r^{pr}_1 \) is a correct implementation of \( r^d_1 \), and \( r^{pr}_2 \) is a correct implementation of \( r^d_2 \), then \( \text{nondet}(r^{pr}_1, r^{pr}_2) \) (i.e., \( (r_1; r_2) \)) is a correct implementation of \( \text{nondet}(r^d_1, r^d_2) \).

Formally, let \( \mathcal{A}^pr \) be the state of the Prolog database, and \( \mathcal{F}(\mathcal{B}^d) = \mathcal{A}^pr \) be its equivalent in the model-level ASM representation of VPM models, and let \( \text{nondet}(r^{pr}_1, r^{pr}_2) \) be the query submitted to the Prolog engine. If the Prolog execution succeeds with a result state \( \mathcal{B}^pr \) (denoted as \( \mathcal{A}^pr \Rightarrow \mathcal{B}^pr \)) then there exists a state \( \mathcal{B}^d \) such that \( \mathcal{B}^pr = \mathcal{F}(\mathcal{B}^d) \) and \( \mathcal{B}^d = \text{next}_{\text{nondet}}(r^d_1, r^d_2)(\mathcal{A}^d) \).

**Proof.** Since Prolog is a deterministic programming language (while graph transformation can be nondeterministic), \( r^d_2 \) can only be called in \( \text{nondet}(r^{pr}_1, r^{pr}_2) \), if \( r^{pr}_1 \) fails. However, this is not a problem, since we only prove the correctness of our approach but not the completeness.

Now, for all \( i = 1, 2 \) if \( r^{pr}_i \) succeeds, then \( r^d_i \) is applicable as well on the same matching. Moreover, since each choicepoint inside \( r^d_i \) is cut off after the first successful application of the rule, the control will never return there for the same matching. \( \Box \)

**Proposition 6.17 (Correctness of sequential application).** If \( r^{pr}_1 \) is a correct implementation of \( r^d_1 \), and \( r^{pr}_2 \) is a correct implementation of \( r^d_2 \), then \( \text{seq}(r^{pr}_1, r^{pr}_2) \) (or, in Prolog syntax, \( r_1, r_2 \)) is a correct implementation of \( \text{seq}(r^d_1, r^d_2) \).

Formally, let \( \mathcal{A}^pr \) be the state of the Prolog database, and \( \mathcal{F}(\mathcal{B}^d) = \mathcal{A}^pr \) be its equivalent in the model-level ASM representation of VPM models, and let \( \text{seq}(r^{pr}_1, r^{pr}_2) \) be the query submitted to the Prolog engine. If the Prolog execution succeeds with a result state \( \mathcal{B}^pr \) (denoted as \( \mathcal{A}^pr \Rightarrow \mathcal{B}^pr \)) then there exists a state \( \mathcal{B}^d \) such that \( \mathcal{B}^pr = \mathcal{F}(\mathcal{B}^d) \) and \( \mathcal{B}^d = \text{next}_{\text{seq}}(r^d_1, r^d_2)(\mathcal{A}^d) \).

**Proof.** A trivial consequence of (i) the equivalence of the Prolog " , " operator and the \( \text{seq} \) operator of ASM, and (ii) the fact that all choicepoints (of Prolog rules) are cut off in the implementation of any other control structures. \( \Box \)

**Proposition 6.18 (Correctness of loop).** If \( r^{pr} \) is a correct implementation of \( r^d \), then \( \text{loop}(r^{pr}) \) is a correct implementation of \( \text{loop}(r^d) \) provided that \( \text{loop}(r^d) \) is terminating.

Formally, let \( \mathcal{A}^pr \) be the state of the Prolog database, and \( \mathcal{F}(\mathcal{B}^d) = \mathcal{A}^pr \) be its equivalent in the model-level ASM representation of VPM models, and let \( \text{loop}(r^{pr}) \) be the query submitted to the Prolog engine. If the Prolog execution succeeds with a result state \( \mathcal{B}^pr \) (denoted as \( \mathcal{A}^pr \Rightarrow \mathcal{B}^pr \)) then there exists a state \( \mathcal{B}^d \) such that \( \mathcal{B}^pr = \mathcal{F}(\mathcal{B}^d) \) and \( \mathcal{B}^d = \text{next}_{\text{loop}}(r^d)(\mathcal{A}^d) \).

**Proof.** If \( r^{pr} \) is successfully applied in \( \text{try} \) mode, then \( r^d \) is also applicable in \( \text{try} \) mode, and it returns with true. On the other hand, if the application of \( r^{pr} \) in \( \text{try} \) mode fails then the execution of \( \text{loop}(r^{pr}) \) terminates. However, in such a case \( \text{try}(r^d) \) returns with false, thus the \( \text{while} \) statement in \( \text{loop}(r^d) \) also terminates.

Since \( r^{pr} \) is applied in \( \text{try} \) mode, all internal choicepoints in \( r^{pr} \) are cut off, thus the control will never return. \( \Box \)
Proposition 6.19 (Correctness of forall). If $r^{pr}$ is a correct implementation of $r^o$, then the rule $forall(r^{pr})$ is a correct implementation of $forall(r^o)$ provided that $r^o$ is independent of itself (i.e., the application of the rule will not disable potential matchings).

Formally, let $A^{pr}$ be the state of the Prolog database, and $F(A^{pr}) = A^{pr}$ be its equivalent in the model level ASM representation of VPM models, and let $forall(r^{pr})$ be the query submitted to the Prolog engine. If the Prolog execution succeeds with a result state $A^{pr}$ (denoted as $A^{pr} = F(A^{pr})$) then there exists a state $A^o$ such that $A^{o} = F(A^{pr})$ and $A^{o} = next_{forall(r^{pr})}(A^{pr})$.

Proof. This is the only case when internal choicepoints of a rule (generated during the pattern matching of the LHS) are of immense importance. Let $p(X)$ be a node $i$ in the LHS of $r^{pr}$ with a generated choicepoint. The first time it is reached by the execution, all resolving candidates $cand(i) = \{p(a), p(b), \ldots\}$ are calculated, and the first candidate $p(a)$ is tried to be unified with $p(X)$. If $r^{pr}$ is successfully applied on a matching, then backtracking is initiated by fail (in the Prolog implementation of $forall$), thus the next resolving candidate $p(b)$ is investigated.

Since $r^o$ is independent of itself by assumption (i.e., the potential matchings are not in conflict), therefore the Prolog computation is consistent, and it traverses all possible combinations instantiating variables in the LHS.

Proposition 6.38. Let $A$, $B$ and $C$ be ASM states of the VPM representation of a finite automaton. Moreover, let $r^o$ denote $forall(\text{initR})$ while $r_{as}$ denote $\text{asInitR}$.

- **Completeness.** For all $B = next_{r^o}(A)$, there exists an execution of the AS (with an automaton $a$ passed as parameter to $r_{as}$) which yields the same state as result, i.e., $\exists C : C = next_{r_{as}}(A) \land B = C$.

- **Correctness.** For all $B = next_{r_{as}}(A)$ where $r_{as}$ is called with an automaton $a$ passed as parameter, there exists an execution of the GTS which yields the same state as result, formally, $\exists C : C = next_{r^o}(A) \land B = C$.

Proof. Since the negative condition, and the model manipulation parts are identical in Alg. 20 and Alg. 21, we only have to show that the there exists a bidirectional mapping between the matching of the LHSs.

Thus, to construct the proof, we establish a bidirectional mapping between the values of variables $\langle A, a, 1 \rangle$, $\langle S, s, 1 \rangle$, $\langle C, c, 1 \rangle$ and $\langle C, c, 2 \rangle$.

Due to the meta-level and model-level representation of VPM models (as discussed in Chapter 3), we can state the following.

- **A1-a1.** Since, for any $a$, $\text{context}(a)$ in Line 1 of Alg. 20) holds if and only if the formula $\text{context}(a)$ in Line 2 of Alg. 21) is true thus the same value $a$ can be assigned to them at any time (where a is passed as a parameter to Alg. 21);

- **S1-s1.** Since, for any $s$, $\text{state}(s)$ in Line 1 of Alg. 20) holds if and only if $\text{context}(s, s)$ in Line 5 of Alg. 21) is true thus the same value $s$ can be assigned to them at any time to this variable pair.

- **S1-s1.** If $\text{context}(c, a, s)$ in Line 1 of Alg. 20) holds for some triple $\langle c, a, s \rangle$, then $a$ and $s$ are entities, thus $s$ is in the collection obtained in Line 3 of Alg. 21, which means that $c1 = c$ (and vice versa, which can be proved with a similar reasoning).

- **C2-c2.** Since both $C2$ and $c2$ are instantiated from the reserve, we may say that the same value is assigned to them at each time.

B.4 Proofs of Chapter 6

Proposition 7.10 (Equivalence of init states). The initial states $A^{pr}_{inil}$ of ASM$^{pr}$ and $A^{mv}_{inil}$ of ASM$^{mv}$ are equivalent with respect to $F_1$. Formally, $\forall A^{pr}_{inil} \exists A^{mv}_{inil} : F_1(A_{inil}^{pr}) = F_1(A_{inil}^{mv})$, and $\forall A_{inil}^{mv} \exists A^{pr}_{inil} : A_{inil}^{mv} = F_1(A_{inil}^{pr})$. 


Proof. A trivial consequence of the construction in Sec. 6.3.2.

**Proposition 7.11.** We can establish a bisimulation between the steps/runs and a bidirectional mapping between states of $ASM^{gr}$ and $ASM^{mw}$ (provided that $nextIdFromReserve$ is implemented correctly). Formally,

1. **Completeness / Forward simulation.**
   \[ \forall \alpha^gr \exists \alpha^mw \forall \beta^gr \exists \beta^mw : \alpha^mw = F_1(\alpha^gr) \land \beta^gr = next_R(\alpha^gr) \rightarrow \exists \beta^mw : \beta^gr = next_{F_1(R)}(\alpha^mw) \land \beta^mw = F_1(\beta^gr). \]

2. **Correctness / Backward simulation.**
   \[ \forall \alpha^gr \exists \alpha^mw \forall \beta^gr \exists \beta^mw : \alpha^mw = F_1(\alpha^gr) \land \beta^gr = next_{F_1(R)}(\alpha^mw) \rightarrow \exists \beta^mw : \beta^gr = next_R(\alpha^gr) \land \beta^mw = F_1(\beta^gr). \]

Proof (Sketch). Each direction is proved separately.

**Completeness**

Let $r^{gr}$ be an enabled rule in $ASM^{gr}$, and $R^{mw}$ be the set of transitions in $ASM^{mw}$ generated from $r^{gr}$ by Alg. 22.

Let $\zeta_{hs} = \{ x_1 \mapsto a_1, \ldots, x_n \mapsto a_n \}$ denote a variable assignment for a successful matching of the rule $r^{gr}$ (i.e., which respects the negative conditions as well). Since $\{a_1, \ldots, a_n\}$ is an element from $dom(x_1) \times \ldots \times dom(x_n)$, thus we executed the loop of Line 2 in Alg. 22 to obtain a transition $\tau \in R^{mw}$ that is enabled exactly for the matching $\zeta_{hs}$. Due to the encoding (Lines 9–11 in Alg. 22), the updates of $\tau$ are identical with the updates of $r^{gr}$ provided that the same locations are accessed. Since $\zeta_{hs}$ is already defined, this only leaves us to prove the non-deterministic creation of new objects is correctly implemented by Lines 6–8 of Alg. 22, which is assumed here for $nextIdFromReserve$.

Since $\alpha^mw = F_1(\alpha^gr)$ by assumption, and the update set is identical, this finishes the proof of completeness.

**Correctness**

Now let $r^{mw}$ be an enabled transition (or, to be precise, rule) in $ASM^{mw}$, and we have to show that it corresponds to a successful application of a rule $r^{gr}$ in $ASM^{gr}$ for some matching $\zeta_{hs}$.

Since $r^{mw}$ is enabled therefore all locations $\{a_1, \ldots, a_n\}$ appearing in its guard $g_{hs}$ should be true (moreover, the negative guard $g_{neg}$ should be satisfied as well). As $\alpha^mw = F_1(\alpha^gr)$ holds by assumption, therefore the truth values of the same locations in $ASM^{gr}$ are identical.

Due to the fact that $r^{mw}$ was generated by Alg. 22, there exists a rule $r^{gr}$ in $ASM^{gr}$ from which it was derived. However, notice that the variable assignment $\zeta_{hs} := \{ x_1 \mapsto a_1, \ldots, x_n \mapsto a_n \}$ satisfies $\phi_{hs}(x_1, \ldots, x_n)$ (while $\phi_{neg}$ is also respected), therefore, rule $r^{gr}$ is enabled as well. Moreover, since $g^{mw}$ was generated from $r^{gr}$ the updates are identical as well (as we already proved for completeness).

Applying the rule on the same matching with the same update set implies the correctness of the rule application, formally $\beta^{mw} = F_1(next_{r^{gr}}(\alpha^{gr}))$. \(\square\)

**Proposition 7.12 (Equivalence of initial states).** The initial states $\alpha^{mw}_{init}$ of $ASM^{mw}$ and $\alpha^{opt}_{init}$ of $ASM^{opt}$ are equivalent with respect to $F_2$. Formally, $\forall \alpha^{mw}_{init} \exists \alpha^{opt}_{init} : \alpha^{opt}_{init} = F_2(\alpha^{mw}_{init})$, and $\forall \alpha^{opt}_{init} \exists \alpha^{mw}_{init} : \alpha^{mw}_{init} = F_2(\alpha^{opt}_{init})$.

Proof (Sketch). Locations derived from dynamic elements are identical in $\alpha^{mw}_{init}$ and $\alpha^{opt}_{init}$ due to the construction of Sec. 6.3.3, while static elements of $\alpha^{mw}_{init}$ are not projected into $\alpha^{opt}_{init}$.

**Proposition 7.13.** We can establish a bisimulation between the steps/runs and a bidirectional mapping between states of $ASM^{mw}$ and $ASM^{opt}$. Formally,
1. **Completeness / Forward simulation.**

\( \forall \mathfrak{A}^\text{mv} \forall \mathfrak{A}^\text{opt} \forall \mathfrak{B}^\text{mv} : \mathfrak{A}^\text{opt} = \mathcal{F}_2(\mathfrak{A}^\text{mv}) \land \mathfrak{B}^\text{mv} = \text{next}_R(\mathfrak{A}^\text{mv}) \rightarrow \exists \mathfrak{B}^\text{opt} : \mathfrak{B}^\text{opt} = \text{next}_\mathcal{F}_2(R)(\mathfrak{A}^\text{opt}) \land \mathfrak{B}^\text{opt} = \mathcal{F}_2(\mathfrak{B}^\text{mv}) \).

2. **Correctness / Backward simulation.**

\( \forall \mathfrak{A}^\text{mv} \forall \mathfrak{A}^\text{opt} \forall \mathfrak{B}^\text{mv} : \mathfrak{A}^\text{opt} = \mathcal{F}_2(\mathfrak{A}^\text{mv}) \land \mathfrak{B}^\text{mv} = \text{next}_\mathcal{F}_2(R)(\mathfrak{A}^\text{mv}) \rightarrow \exists \mathfrak{B}^\text{mv} : \mathfrak{A}^\text{mv} = \text{next}_R(\mathfrak{A}^\text{mv}) \land \mathfrak{B}^\text{opt} = \mathcal{F}_2(\mathfrak{B}^\text{mv}) \).

**Proof (Sketch).** Each direction is proved separately (using the notation of Alg. 23).

**Completeness**

Let \( \tau^\text{mv} = g^\text{mv}_{\text{hs}} \land g^\text{mv}_{\text{neg}} \rightarrow \text{act}^\text{mv} \) be an enabled transition (rule) in \( \text{ASM}^\text{mv} \). Let \( \tau^\text{opt} = g^\text{opt}_{\text{hs}} \land g^\text{opt}_{\text{neg}} \rightarrow \text{act}^\text{opt} \) be a transition in \( \text{ASM}^\text{opt} \) generated from \( \tau^\text{mv} \) as a result of Alg. 23. We first show that \( \tau^\text{opt} \) is enabled.

Since \( \tau^\text{mv} \) is enabled, therefore all (positive) literals \( p_i \) in \( g^\text{mv}_{\text{hs}} \) accessing locations in \( \text{ASM}^\text{mv} \) are evaluated to true in state \( \mathfrak{A}^\text{mv} \) (\( \forall i : [p_i]^\mathfrak{A}^\text{mv} = \text{true} \)). As a consequence, we can state the following.

- If \( p_i \) is static, then \( p_i \) does not appear in the guard \( g^\text{opt}_{\text{hs}} \) of \( \tau^\text{opt} \).
- If \( p_i \) is dynamic, then \( p_i \) appears in the guard \( g^\text{opt}_{\text{hs}} \), but \( [p_i]^\mathfrak{A}^\text{opt} = \text{true} \).

Furthermore, as \( \tau^\text{mv} \) is enabled, none of the (negated) clauses \( \neg \phi_i \) in \( g^\text{mv}_{\text{neg}} \) are violated in state \( \mathfrak{A}^\text{mv} \) (i.e., \( \forall i : [\neg \phi_i]^\mathfrak{A}^\text{mv} = \text{true} \)). Therefore, for all negated clause \( \phi_i \) there exists a literal \( p_{i,j} \) for which \( [p_{i,j}]^\mathfrak{A}^\text{opt} = \text{false} \).

- If \( p_{i,j} \) is static then \( \phi_i \) does not appear in \( g^\text{opt}_{\text{neg}} \).
- If \( p_{i,j} \) is dynamic, then it appears in the guard \( g^\text{opt}_{\text{neg}} \), but \( [p_{i,j}]^\mathfrak{A}^\text{opt} = \text{false} \) implying that \( [\neg \phi_i]^\mathfrak{A}^\text{opt} = \text{true} \).

Therefore \( \tau^\text{opt} \) is enabled whenever \( \tau^\text{mv} \) is enabled. As \( \text{act}^\text{mv} \) is identical with \( \text{act}^\text{opt} \), the same locations are updated in both cases. Hence \( \mathfrak{B}^\text{opt} = \text{next}_\mathcal{F}_2(R)(\mathfrak{A}^\text{opt}) = \mathcal{F}_2(\mathfrak{B}^\text{mv}) \), which finishes the proof of completeness.

**Correctness**

Let \( \tau^\text{opt} = g^\text{opt}_{\text{hs}} \land g^\text{opt}_{\text{neg}} \rightarrow \text{act}^\text{opt} \) be an enabled transition in \( \text{ASM}^\text{opt} \) generated from some transition \( \tau^\text{mv} = g^\text{mv}_{\text{hs}} \land g^\text{mv}_{\text{neg}} \rightarrow \text{act}^\text{mv} \) in \( \text{ASM}^\text{mv} \) as a result of Alg. 23. Again as \( \tau^\text{opt} = \tau^\text{mv} \), we only have to show that \( \tau^\text{opt} \) is enabled whenever \( \tau^\text{opt} \) is enabled.

Since \( \tau^\text{opt} \) is enabled, therefore all (positive) literals \( p_i \) in \( g^\text{opt}_{\text{hs}} \) accessing locations in \( \text{ASM}^\text{opt} \) are evaluated to true in state \( \mathfrak{A}^\text{opt} \) (i.e., \( \forall i : [p_i]^\mathfrak{A}^\text{opt} = \text{true} \)). Let us now suppose by contradiction that \( \tau^\text{mv} \) is not enabled because there exists a literal \( p_j \) for which \( [p_j]^\mathfrak{A}^\text{mv} = \text{false} \) in the original (naive) transition system.

- If \( p_j \) is static, then \( \tau^\text{mv} \) would be eliminated (Lines 5–6 in Alg. 23), which contradicts the fact that \( \tau^\text{opt} \) is generated from \( \tau^\text{mv} \).
- If \( p_j \) is dynamic, then \( p_j \) appears in the guard \( g^\text{opt}_{\text{hs}} \), but \( [p_j]^\mathfrak{A}^\text{opt} = \text{true} \) and \( [p_j]^\mathfrak{A}^\text{opt} = \text{false} \), which is a contradiction as well.

Furthermore, as \( \tau^\text{opt} \) is enabled, none of the (negated) clauses \( \neg \phi_i^\text{opt} \) in \( g^\text{opt}_{\text{neg}} \) are violated in state \( \mathfrak{A}^\text{opt} \) (i.e., \( \forall i : [\neg \phi_i^\text{opt}]^\mathfrak{A}^\text{opt} = \text{true} \)). Therefore, for all \( \phi_i^\text{opt} \) there exists a literal \( p_{i,j,0} \) for which \( [p_{i,j,0}]^\mathfrak{A}^\text{opt} = \text{false} \).

Let us now suppose by contradiction that \( \tau^\text{mv} \) is not enabled because there exists a clause \( \phi_0^\text{mv} \) in the original (naive) transition system \( \tau^\text{mv} \) for which \( [\neg \phi_0^\text{mv}]^\mathfrak{A}^\text{mv} = \text{false} \). This may happen only if \( \forall j : [p_{i,j}]^\mathfrak{A}^\text{mv} = \text{true} \) where \( \phi_0^\text{mv} \equiv \bigwedge_j p_{i,j} \).
• Now if all $p_{i_0,j}$ is static, then $\tau^{mv}$ would be eliminated (Lines 14–15 in Alg. 23), which contradicts the fact that $\tau^{opt}$ is generated from $\tau^{mv}$.

• If there is a dynamic literal $p_{i_0,j}$ in $\phi_0^{mv}$, then this literal also appears in the guard $g_{neg}^{opt}$ in the corresponding clause $\phi_0^{opt}$. This causes a contradiction, since there exists a literal $p_{i_0,j_0}$ for which $\llbracket p_{i_0,j_0} \rrbracket^{opt} = false$ but $\llbracket p_{i_0,j_0} \rrbracket^{mv} = true$.

As a result, we have $\mathfrak{B}^{opt} = \mathcal{F}_2(\mathfrak{B}^{mv})$ where $\mathfrak{B}^{mv} = \text{next}_R(\mathfrak{A}^{mv})$, which completes the proof of correctness.
C

SAL Descriptions of the Dining Philosophers Case Study

C.1 SAL Description for the Model-Level Encoding of Dining Philosophers

Note that only the behavior of philosopher phi is depicted there due to the symmetric nature of the problem to improve clarity.

ForkID : TYPE = {f1,f2,f3};
PhiID : TYPE = {ph1,ph2,ph3};
StatusRng : TYPE = {think, hungry, hasL, eat, hasR};
dinphi_gt : MODULE =
BEGIN
  GLOBAL hold: ARRAY PhiID OF ARRAY ForkID OF BOOLEAN
  GLOBAL status: ARRAY PhiID OF StatusRng
  INITIALIZATION
    status[ph1] = think;
    hold[ph1][f1] = FALSE;
    hold[ph1][f2] = FALSE;
    hold[ph1][f3] = FALSE;
  TRANSITION
    status[ph1] = think --> status[ph1] = hungry; []
    status[ph1] = hungry AND NOT hold[ph2][f1] -->
      hold[ph1][f1] = TRUE;
      status[ph1] = hasL; []
    status[ph1] = hasL AND NOT hold[ph3][f3] -->
      hold[ph1][f3] = TRUE;
      status[ph1] = eat; []
    status[ph1] = eat -->
      hold[ph1][f1] = FALSE;
      status[ph1] = hasR; []
    status[ph1] = hasR -->
      hold[ph1][f3] = FALSE;
      status[ph1] = think;
END;

C.2 SAL Description of the Meta-Level Encoding of Dining Philosophers

The meta-level encoding of a single dining philosopher is listed below in the SAL format.

Conf_dom : TYPE = { think, hungry, hasLeft, hasRight, eat, free, held};
Step_dom : TYPE = {t1, t2, t3, t4, t5, t6, t7};
Event_dom : TYPE = { acq, rel };
Aut_dom : TYPE = { ph1, ph2, ph3, f1, f2, f3};
pc_rng : TYPE = {fireR, addQR};
dinphi_sc_gt : MODULE =
BEGIN
  GLOBAL isAct : ARRAY [Aut_dom] of ARRAY [Conf_dom] of BOOLEAN;
  ...
GLOBAL inQueue : ARRAY [Aut_dom] of ARRAY [Event_dom] of BOOLEAN;
GLOBAL fire : ARRAY [Aut_dom] of ARRAY [Step_dom] of BOOLEAN;
GLOBAL pc : ARRAY [Aut_dom] of pc_rng;

INITIALIZATION
isAct[ph1][think] = TRUE;    isAct[ph1][hungry] = FALSE;
isAct[ph1][hasLeft] = FALSE;  isAct[ph1][eat] = FALSE;
isAct[ph1][hasRight] = FALSE;  isAct[ph1][free] = FALSE;
isAct[ph1][held] = FALSE;
inQueue[ph1][acq] = FALSE;  inQueue[ph1][rel] = FALSE;
fire[ph1][t1] = FALSE;  fire[ph1][t2] = FALSE;  fire[ph1][t3] = FALSE;
fire[ph1][t6] = FALSE;
pc[ph1] = fireR;

TRANSITION
% fireNoEvtNoInStr
pc[ph1] = fireR AND isAct[ph1][think]) AND NOT fire[ph1][t1] -->
  isAct[ph1][think] = FALSE;
  isAct[ph1][hungry] := TRUE;
  pc[ph1] := addQR;
  fire[ph1][t1] := TRUE; [ ]

% fireNoEvtWithInStr
pc[ph1] = fireR AND isAct[ph1][hungry]) AND isAct[f1][free]
AND NOT fire[ph1][t2] -->
  isAct[ph1][hungry] := FALSE;
  isAct[ph1][hasLeft] = TRUE;
  pc[ph1] := addQR;
  fire[ph1][t2] := TRUE; [ ]

% fireNoEvtWithInStr
pc[ph1] = fireR AND isAct[ph1][hasLeft]) AND isAct[f3][free]
AND NOT fire[ph1][t3] -->
  isAct[ph1][hasLeft] := FALSE;
  isAct[ph1][eat] = TRUE;
  pc[ph1] := addQR;
  fire[ph1][t3] := TRUE; [ ]

% fireNoEvtNoInStr
pc[ph1] = fireR AND isAct[ph1][eat]) AND NOT fire[ph1][t4] -->
  isAct[ph1][eat] := FALSE;
  isAct[ph1][hasRight] = TRUE;
  pc[ph1] := addQR;
  fire[ph1][t4] := TRUE; [ ]

% fireNoEvtNoInStr
pc[ph1] = fireR AND isAct[ph1][eat]) AND NOT fire[ph1][t5] -->
  isAct[ph1][hasRight] := FALSE;
  isAct[ph1][think] = TRUE;
  pc[ph1] := addQR;
  fire[ph1][t5] := TRUE; [ ]

% addQueueNoActR
pc[ph1] = addQR AND fire[ph1][t1] -->
  pc[ph1] := fireR;
  fire[ph1][t1] := FALSE; [ ]

% addQueueWithActR
pc[ph1] = addQR AND fire[ph1][t2] -->
  pc[ph1] := fireR;
  inQueue[t2][acq] := TRUE;
  fire[ph1][t2] := FALSE; [ ]

% addQueueWithActR
pc[ph1] = addQR AND fire[ph1][t3] -->
  pc[ph1] := fireR;
  inQueue[t3][acq] := TRUE;
  fire[ph1][t3] := FALSE; [ ]

% addQueueWithActR
pc[ph1] = addQR AND fire[ph1][t4] -->

C.3 SAL Descriptions for Model-level Encoding of Philosophers’ Statecharts

This description was yielded by adapting the techniques described in [105] to the SAL and Murō systems. Only the transitions of a single philosopher are included this time as well.

```sal
BEGIN
  hold : array [PhilID] of array [ForkID] of boolean;
  ph_status : array [PhilID] of phil_status_rng;
  f_status : array [ForkID] of fork_status_rng;
  f_queue : array [ForkID] of array [Event_dom] of boolean;

  INITIALIZE
    hold[phi1][f1] = FALSE; hold[phi1][f2] = FALSE; hold[phi1][f3] = FALSE;
    ph_status[phi1] = think; ...

  TRANSITION
    ph_status[phi1] = think --\> ph_status[phi1] = hungry; []
    ph_status[phi1] = hungry AND f_status[f1] = free
      AND NOT hold[phi2][f1] --\> hold[phi1][f1] = TRUE;
    f_queue[f1][acq] := true;
    ph_status[phi1] = hasL; []
    ph_status[phi1] = hasL AND f_status[f3] = free
      AND NOT hold[phi3][f3] --\> hold[phi1][f3] = TRUE;
    f_queue[f3][acq] := true;
    ph_status[phi1] = eat; []
    status[phi1] = eat --\> hold[phi1][f1] = FALSE;
    f_queue[f1][rel] := true;
    status[phi1] = hasR; []
    status[phi1] = hasR --\> hold[phi1][f3] = FALSE;
    f_queue[f3][rel] := true;
    status[phi1] = think;
END;
```

C.3 SAL Descriptions for Model-level Encoding of the UML Statecharts for Dining Philosophers
References

References


References


References


