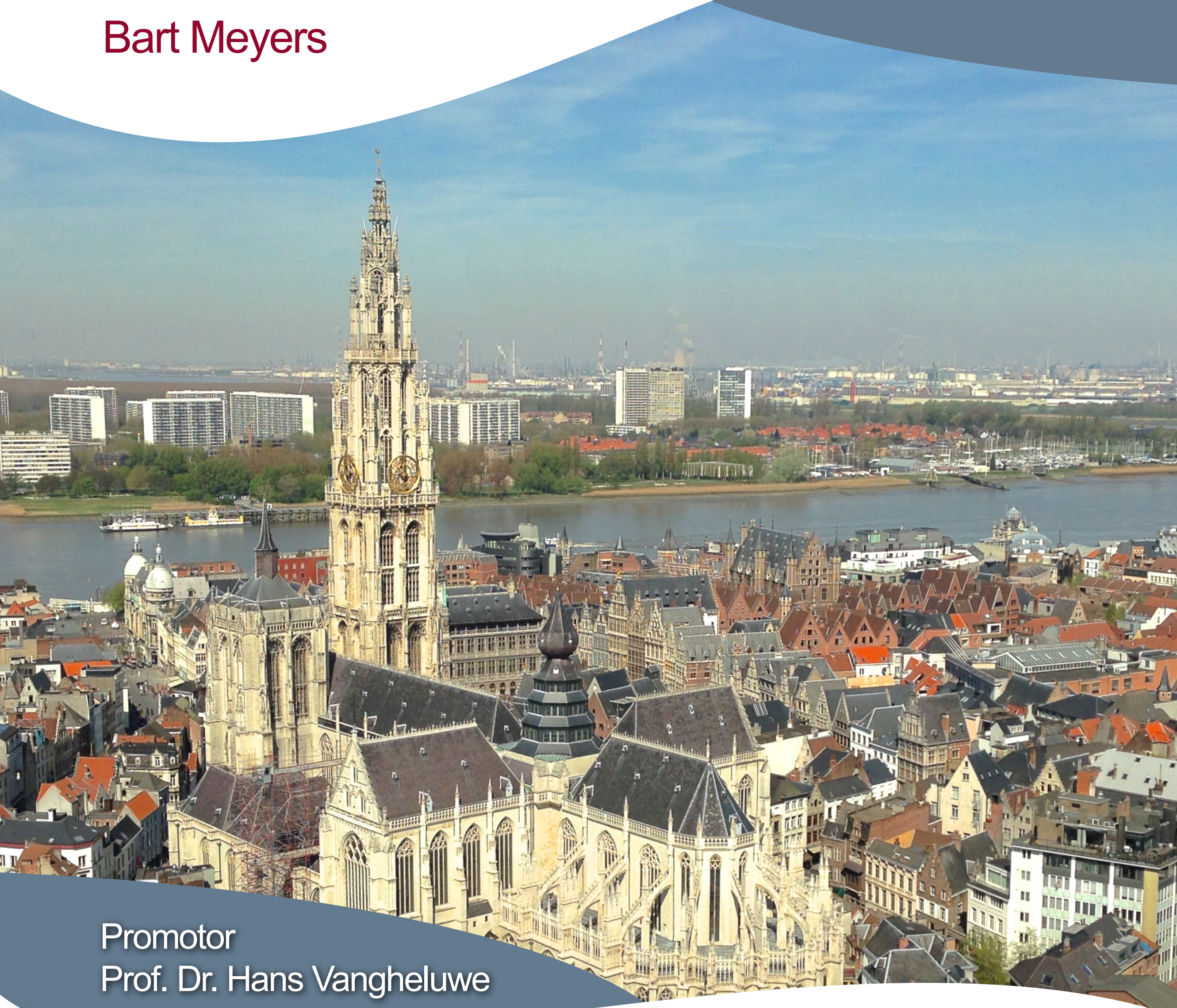
A Multi-Paradigm Modelling Approach to Design and Evolution of Domain-Specific Modelling Languages

Proefschrift voorgelegd tot het behalen van de
graad van Doctor in de Wetenschappen: Informatica
aan de Universiteit Antwerpen te verdedigen door:

Bart Meyers



Promotor
Prof. Dr. Hans Vangheluwe

Faculteit Wetenschappen
Departement Wiskunde-Informatica

Antwerpen, 2016

Universiteit
Antwerpen

Acknowledgments

Bart Meyers
Antwerpen, February 2016

I would like to thank Hans for introducing me to modelling and its community. Hans, you have taught me so many things, both on topic and off topic. I cannot imagine a better advisor. I am looking forward to continue working together.

Thanks to the members of the jury of this thesis for their remarks and insights. Antonio, Dirk, Juan, Serge and Tom, thank you for your time – I enjoyed our discussions.

I would like to thank all members of our research group, past and present, for all (long distance) discussions. Ahmed, Alessandro, Ali, Anna, Anne, Bart, Bentley, Bruno, Cláudio, Diana, Dirk, Eugene, Frederik, Hans, István, Jan, Javier, Ken, Levi, Margarete, Maris, Naomi, Olaf, Peter, Pieter, Quinten, Raphael, Sadaf, Serge, Simon, Tim, Xiaobo and Yentl, thank you for your input, discussions, good times, and (extended) lunches.

I was fortunate to collaborate regularly with some of the brightest minds on this planet during research stays and workshops. Juan, thank you for inviting me to Madrid and introducing me to *metaDepth*. Manuel, and Gerti, thank you for inviting me to Vienna. It was a pleasure to work with you. Frédéric, Cécile and Christophe, thank you to invite us to Paris, and showing me around in the research area of simulation. Thank you, Antonio, Eugene, Jonathan, Levi, Manuel, Pieter and Romuald, for the fruitful discussions near the beach. Hans and Pieter, thank you for organising these CaMPAM workshops. You are all more than welcome in Antwerp!

Thank you to my family and friends, who took my mind off things when I needed it. Last but not least, thank you, Sofie, for all your love and support, especially in the last few weeks of writing my thesis. I can't wait to go on holiday with you.

Abstract

A recent approach to tackle the ever increasing complexity of software intensive systems is Model-Driven Engineering (MDE) [182]. Development of such complex systems spans a plethora of different activities, including requirements modelling, analysis, design, testing, etc. for often very diverse system components and views. This inherent heterogeneity is tackled in *Multi-Paradigm Modelling* (MPM) by explicitly modelling all aspects of the system under study at the most appropriate level(s) of abstraction using the most appropriate modelling language(s) [146, 45]. This includes the explicit modelling of the development processes. In Domain-Specific Modelling (DSM) [74] the general goal is to provide means for domain users to model systems using concepts and notations they are familiar with, in their problem domain. Techniques such as metamodelling and model transformation enable modelling language engineers and domain experts to create Domain-Specific Modelling Languages (DSMLs) for the domain users. Because syntax and semantics of DSMLs are precisely defined by means of metamodelling and model transformation, models can be used for analysis, simulation, optimisation, documentation and even full code synthesis. According to MPM principles, multiple aspects/views as well as sub-systems are modelled using distinct, most appropriate DSMLs. This means that not only system modelling, but also modelling language engineering, becomes part of the development process. Language engineering is thus a vital part of DSM. This thesis presents contributions in three research areas of language engineering: verification support for DSMLs, evolution of DSMLs and composition of DSMLs. An elevator control system is used as a running example throughout the thesis.

Verification support for DSMLs. Verifying whether a model satisfies a set of requirements is considered to be an important challenge in DSM [65]. It is nevertheless mostly neglected by current DSM approaches. We present a solution in the form of *ProMoBox*, a framework that integrates the definition and verification of temporal properties in discrete-time behavioural DSMLs, whose semantics can be described as a schedule of graph rewrite rules. Thanks to the expressiveness of graph rewriting, this covers a very large class of problems. With *ProMoBox*, the domain user models not only the system with a DSML, but also its properties, input model, run-time state and output trace. A DSML is thus comprised of five sublanguages, which share domain-specific syntax. The sublanguages are generated from a single metamodel, that is annotated to denote the role of each language concept. The operational semantics of the DSML is modelled as a transformation and is annotated with information about input and output. The modelled system and its properties are translated to *Promela*, and properties are verified with *Spin*, a tool for explicit state model checking [89]. In case a counterexample is found, its execution trace is transformed to the domain-specific level as a trace model, which can be played out. Thus, whilst modelling and verifying properties, the domain user is shielded from underlying notations and techniques. Following MPM principles, we explicitly model the *ProMoBox* framework's process in a Formalism Transformation

Graph and Process Model [125]. Furthermore, we evaluate *ProMoBox* to assert that it supports the specification and verification of properties in a highly flexible and automated way, according to MPM principles.

Evolution of DSMLs. In software engineering, the evolution of software artifacts is ubiquitous [131]. In DSM, where modelling languages play a central role, evolution occurs not only at the level of models, but also at the level of modelling languages [180]. Consequently, support for (semi-)automated co-evolution of instance models or transformation models is needed. We present a framework that deals with all possible consequences of language evolution in a DSML relational ecosystem. We identify consistency and continuity as a goal, meaning that the evolution must be syntactically correct (*i.e.,* the conformance relation is preserved throughout the system) and semantically correct (*i.e.,* the system has evolved according to the intended changes). Contrary to related approaches at the time of this work, we focus on the neglected problem of transformation co-evolution. We de-construct all possible consequences of evolution in the context of a DSML relational ecosystem and identify four basic evolution scenarios: model evolution, image evolution, domain evolution and transformation evolution, which can each be handled (semi-)automatically. Based on this we define and implement an approach for instance co-evolution and transformation co-evolution that uses a flexible and modular migration pipeline. Furthermore, we present a feature diagram of all evolution consequences and an algorithm that forms the backbone of our framework, describing all possible co-evolution scenarios in detail.

Composition of DSMLs. Current methods for the development of DSMLs require the language engineer to develop DSMLs from scratch. Nevertheless, there is a clear need for reuse of existing languages, or language modules [159]. We use three composition mechanisms from the *metaDepth* tool, namely, extension, concepts and templates and investigate whether they are sufficient to allow composition of modelling languages in a structured way. All three concepts, abstract syntax, (textual) concrete syntax and (operational) semantics, of a modelling language are combined using these mechanisms. By using these mechanisms, we aim for general applicability of our approach, provided that these composition mechanisms are supported. We conclude however, that only simple composition of semantics can be modelled in a structured way. However, the combination of the operational semantics of languages can be very complex, but necessary in Multi-Paradigm Modelling. Hence, heterogeneous systems must be executed correctly, while still maintaining modularity of languages, to enable reuse, and to allow the combination of languages as black boxes. A solution is to define *semantic adaptation* [23] of heterogeneous parts of a model, which has to be explicitly described. Boulanger *et al.* developed *ModHel'X*, a tool for heterogeneous modelling and semantic adaptation. In *ModHel'X*, complex semantic adaptations have to be coded however, which is tedious and error prone. We introduce a DSML with a visual concrete syntax for modelling such heterogeneous systems. Subsequently, we define a second DSML with textual syntax to model semantic adaptation. This DSML allows users to specify the adaptation of data, control and time using a set of rules. Both DSMLs are translated to the *ModHel'X* framework.

All solutions presented in this thesis adhere to MPM principles: everything is modelled explicitly, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s). As users are exposed to such formalisms, they are shielded from "accidental" complexity by using generative techniques and automation. The solutions in this thesis illustrate the *tool builder*'s approach, as contributions in these thesis are supported by prototype implementations.

Nederlandstalige Samenvatting

Een recente aanpak voor de voortdurende toeneming aan complexiteit in software-intensieve systemen is modelgedreven ontwikkeling (MDE) [182]. De ontwikkeling van zulke complexe systemen houdt een groot aantal verschillende activiteiten in, waaronder het modelleren van vereisten, analyse, ontwerp, testen, enz. voor vaak erg verschillende componenten en facetten van een systeem. *Multi-Paradigmatisch Modelleren* (MPM) probeert een oplossing te bieden voor deze inherente heterogeniteit door alle aspecten van het systeem expliciet te modelleren, op het (de) meest geschikte abstractieniveau(s), in het (de) meest geschikte formalisme(s) [146, 45]. Ook ontwikkelingsprocessen worden expliciet gemodelleerd. Het belangrijkste doel van Domeinspecifiek Modelleren (DSM) [74] is om domeingebruikers toe te laten om systemen te modelleren waarbij ze gebruik maken van concepten en notaties uit hun domein, die hen bekend zijn. Met technieken zoals metamodelleren en modeltransformatie kunnen taalontwikkelaars en domeinexperts domeinspecifieke talen (DSMLs) voor domeingebruikers ontwikkelen. Omdat syntax en semantiek van DSMLs precies gedefinieerd zijn door gebruik te maken van metamodelleren en modeltransformatie kunnen modellen gebruikt worden voor analyse, simulatie, optimalisatie, documentatie en zelfs volledige codegeneratie. Volgens de principes van MPM worden verschillende aspecten, facetten en subsystemen gemodelleerd in verschillende (meest geschikte) DSMLs. Dit betekent dat niet alleen het modelleren van het systeem, maar ook het ontwikkelen van de DSML deel uitmaakt van het ontwikkelingsproces. Taalontwikkeling is dus een essentieel deel van DSM. In deze thesis worden drie contributies in het onderzoeksdomein van taalontwikkeling behandeld, namelijk ondersteuning voor verificatie voor DSMLs, evolutie van DSMLs en samenstelling van DSMLs. Een controlesysteem voor een lift wordt doorheen de thesis gebruikt als voorbeeld.

Ondersteuning voor verificatie voor DSMLs. Verifiëren of een model voldoet aan de vereisten wordt beschouwd als een belangrijke uitdaging in DSM [65]. Desondanks wordt het niet vaak ondersteund in bestaande DSM-oplossingen. We presenteren een oplossing genaamd *ProMoBox*: een raamwerk dat de omschrijving en verificatie van temporele eigenschappen integreert in DSMLs met gedrag in discrete tijd, waarvan de semantiek beschreven kan worden als een structuur met herschrijfregels voor grafen. Omwille van de expressiviteit van het herschrijven van grafen beslaat de oplossing een erg grote klasse van problemen. Met *ProMoBox* modelleert de domeingebruiker niet alleen het ontwerp van systeem met een DSML, maar ook de eigenschappen, het invoermodel, de run-time staat en de uitvoer. Een DSML bestaat dus uit vijf subtalen, die een domeinspecifieke syntax delen. De subtalen worden gegenereerd uit één enkel metamodel, dat geannoteerd wordt om de rol van elk taalconcept vast te leggen. De operationele semantiek van de DSML wordt als een transformatie gemodelleerd en wordt geannoteerd met informatie over invoer en uitvoer. Het gemodelleerde systeem en zijn eigenschappen worden vertaald naar

Promela, en eigenschappen worden geverifieerd met *Spin*, een programma voor model checking met expliciete staat [89]. Als er een tegenvoorbeeld wordt gevonden, dan wordt de uitvoer van het systeem vertaald naar de domeinspecifieke laag als een model dat afgespeeld kan worden. Dit betekent dat de domeingebruiker tijdens het modelleren en verifiëren van eigenschappen wordt afgeschermd van onderliggende notaties en technieken. Volgens de principes van MPM modelleren we het proces van *ProMoBox* in een formalismetransformatiegraaf en procesmodel [125]. Verder evalueren we *ProMoBox* en concluderen we dat het de specificatie en verificatie van eigenschappen ondersteunt op een erg flexibele en geautomatiseerde manier, eveneens volgens de principes van MPM.

Evolutie van DSMLs. In softwareontwikkeling is de evolutie van softwareartifacten alomtegenwoordig [131]. In DSM, waar modelleertalen een centrale rol spelen, komt evolutie niet alleen voor op het niveau van modellen, maar ook van modelleertalen [180]. Er is dan ook nood aan (semi-)geautomatiseerde ondersteuning voor co-evolutie van instantiemodellen of transformatiemodellen. We presenteren een raamwerk dat alle mogelijke gevolgen van taalevolutie in een DSML relationeel ecosysteem behandelt. We definiëren de doelen consistentie en continuïteit, wat betekent dat de evolutie syntactisch correct moet zijn (de conformantierelatie wordt gerespecteerd in het systeem) en semantisch correct (het systeem is geëvolueerd volgens bedoelde veranderingen). In tegenstelling tot ander onderzoek concentreren we ons op het grotendeels genegeerde onderwerp van transformatie co-evolutie. We deconstrueren alle mogelijke consequenties van evolutie in de context van een DSML relationeel ecosysteem en we identificeren vier basisscenarios voor evolutie: modelevolutie, beeldevolutie, domeinevolutie en transformatie-evolutie, die elk een (semi-)automatische oplossing krijgen. Deze oplossingen gebruiken we om een aanpak te definiëren en te implementeren voor de co-evolutie van instanties en transformaties, via een flexibele en modulaire pipeline. Vervolgens presenteren we een featurediagram van alle gevolgen van evolutie en een algoritme dat de ruggegraat vormt van ons raamwerk, dat alle mogelijke co-evolutiescenarios in detail beschrijft.

Samenstelling van DSMLs. Gewoonlijk starten de taalontwikkelaars van nul af aan bij ontwikkeling van DSMLs. Toch is er een duidelijke nood aan hergebruik van bestaande talen of taalmodules [159]. We gebruiken drie compositiemechanismen van de *metaDepth*-tool, namelijk extensie, concepten en sjablonen, en onderzoeken of ze geschikt zijn om op een gestructureerde manier modelleertalen samen te stellen. Elk van de drie facetten – abstracte syntax, (tekstuele) concrete syntax en (operationele) semantiek – van een modelleertaal wordt gecombineerd met deze mechanismen. Door deze mechanismen te gebruiken, mikken we op een generieke toepasbaarheid van onze aanpak, onder de voorwaarde dat deze mechanismen ondersteund worden. We besluiten echter dat enkel eenvoudige semantische samenstellingen op een gestructureerde manier gemodelleerd kunnen worden. Niettemin kan de samenstelling van de operationele semantiek van talen erg complex, maar ook erg nuttig zijn in MPM. Heterogene systemen moeten correct uitgevoerd kunnen worden, maar terwijl moet de modulariteit van de samengestelde talen behouden blijven, om zo hergebruik en het samenstellen van talen als zwarte dozen toe te laten. Een oplossing is om expliciet een *semantische adaptatie* [23] te definiëren tussen heterogene delen van een model. Boulanger *et al.* ontwikkelden *ModHel'X*, een tool voor heterogene modellering en semantische adaptatie. In *ModHel'X* moeten semantische adaptaties echter gecodeerd worden, wat tijdsintensief is en gevoelig voor fouten. Wij presenteren een DSML met visuele concrete syntax voor het modelleren van zulke heterogene systemen. Vervolgens definiëren we een tweede DSML met tekstuele concrete syntax voor het modelleren van semantische adaptatie. Deze DSML laat gebruikers toe om een data-, bestuurs- en tijdsadaptatie te specificeren als een groep van regels. Beide DSMLs worden vertaald naar het *ModHel'X* raamwerk.

Alle oplossingen in de thesis volgen de principes van MPM: alles is expliciet gemodelleerd, op het meest geschikte abstractieniveau, in het meest geschikte formalisme. De gebruikers van deze formalismen worden afgeschermd van “incidentele” complexiteit omdat we gebruik maken van generatieve technieken en automatisering. De oplossingen in deze thesis illustreren de *toolbouwer*-benadering, omdat de contributies ondersteund worden door implementatieprototypes.

Publications

Papers Covered in this Thesis

1. *Bart Meyers and Hans Vangheluwe. A Multi-Paradigm Modelling Approach for the Engineering of Modelling Languages. In "Proceedings of the Doctoral Symposium of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems", CEUR Workshop Proceedings, vol. 1321, p. 1-8, 2014.* Bart and Hans came up with the ideas, Bart wrote the paper and Hans reviewed the paper.

2. *Bart Meyers, Manuel Wimmer, and Hans Vangheluwe. Towards Domain-specific Property Languages: The ProMoBox Approach. In "Proceedings of the 2013 ACM Workshop on Domain-specific Modeling", p. 39-44, ACM New York, NY, USA, 2013.*
Discussions between Bart and Hans resulted in the original idea, Bart and Manuel elaborated the idea, Bart implemented the approach, Bart and Manuel wrote the paper and Hans reviewed the paper.

3. *Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Manuel Wimmer and Hans Vangheluwe. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In "Proceedings of the 7th International Conference on Software Languages Engineering (SLE 2014)", Lecture Notes on Computer Science, vol. 8706, p. 1-20, 2014.*
Bart came up with the original idea in discussions with Hans, the idea was further elaborated by Bart, Romuald, Levi, Eugene and Manuel, Bart implemented the approach, Bart, Romuald, Levi, Eugene and Manuel wrote the paper and Hans reviewed the paper.

4. *Bart Meyers and Hans Vangheluwe. Modelling Language Engineering to Include Temporal Properties in Domain-Specific Modelling. Paper submitted to IEEE Transactions on Software Engineering.*
Bart came up with the original idea in discussions with Hans, Bart elaborated the idea, implemented the approach and wrote the paper, and Hans reviewed the paper.

5. *Bart Meyers and Hans Vangheluwe. Evolution of modelling languages. In Pieter Van Gorp, editor, Fujaba Days 2009: proceedings of the seventh international Fujaba days, pages 29 - 33. Technische Universiteit Eindhoven, November 2009. Eindhoven, The Netherlands.*
The idea came from discussions between Bart and Hans, and they wrote the paper together.

6. *Bart Meyers and Hans Vangheluwe. Evolution of modelling languages. In The eighth BElgian-NEtherlands software eVOLution seminar (BENEVOL), pages 43 - 47, December*

2009. Universite catholique de Louvain, Belgium.
The idea came from discussions between Bart and Hans, and they wrote the paper together.

7. *Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. Science of Computer Programming, 76(12):1223-1246, Elsevier North-Holland, Inc. Amsterdam, The Netherlands, December 2011.*
   The idea came from discussions between Bart and Hans, and they wrote the paper together.

8. *Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. In "Electronic Communications of the European Association of Software Science and Technology (EASST)", 42:1-13, 2012.*
   The original idea came from Bart, the idea was elaborated by Bart, Manuel, Antonio and Jonathan, who all contributed to the implementation of the approach, and writing of the paper.

9. *Bart Meyers, Antonio Cicchetti, Esther Guerra, Juan de Lara. Composing Textual Modelling Languages in Practice. In "6th International Workshop on Multi-Paradigm Modeling (MPM'12)", p. 31-36, ACM New York, NY, USA, 2012.*
   Discussions between Bart, Antonio, Esther and Juan resulted in the original idea, Bart and Juan elaborated the idea, Bart implemented the approach, Bart and Juan wrote the paper and Esther reviewed the paper.

10. *Bart Meyers, Joachim Denil, Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Hans Vangheluwe. A DSL for Explicit Semantic Adaptation. In "7th International Workshop on Multi-Paradigm Modeling (MPM'13)", CEUR Workshop Proceedings, vol. 1112, p. 47-56, 2013.*
    Frédéric and Hans came up with the original idea, Bart, Joachim, Frédéric, Cécile and Christophe elaborated the idea, Bart and Joachim implemented the approach, Bart, Joachim, Frédéric, Cécile and Christophe wrote the paper and Hans reviewed the paper.

Other Papers Published During the Course of this PhD

11. *Manuel Wimmer, Antonio Cicchetti and Bart Meyers. Abstract and Concrete Syntax Migration of Instance Models. In "Proceedings of the ICMT 2010 Transformation Tool Contest", published digitally, 2010.* Manuel, Antonio and Bart came up with the original idea, implemented the idea and wrote the paper together.

12. *Bart Meyers, Peter Ebraert and Dirk Janssens. Intensional changes avoid co-evolution! In "Proceedings of the 7th ECOOP'2010 Workshop on Reflection, AOP and Meta-Data for Software Evolution", p. 4:1–4:6, ACM, New York, NY, USA, 2010.* Peter presented the idea, Bart and Peter elaborated the idea and wrote the paper, Dirk reviewed the paper.

13. *Tim Molderez, Bart Meyers, Dirk Janssens, Hans Vangheluwe. Towards an Aspect-oriented Language Module: Aspects for Petri Nets. In "Proceedings of the seventh workshop on*

Domain-Specific Aspect Languages", p. 21-26, ACM New York, NY, USA, 2012. Hans and Tim came up with the idea, Tim, Bart, Dirk and Hans elaborated the idea, Tim and Bart wrote the paper, and Dirk and Hans reviewed the paper.

14. *Levi Lucio, Sadaf Mustafiz, Joachim Denil, Bart Meyers, and Hans Vangheluwe. The Formalism Transformation Graph as a Guide to Model Driven Engineering. Technical Report SOCS-TR-2012.1, McGill University, March 2012.* Hans, Levi, Sadaf and Joachim came up with the idea, Levi, Sadaf, Joachim, Bart and Hans elaborated the idea, Levi and Sadaf implemented the idea, Levi, Sadaf and Joachim wrote the report and Hans reviewed the report.

15. *Romuald Deshayes, Bart Meyers, Tom Mens and Hans Vangheluwe. ProMoBox in Practice : A Case Study on the GISMO Domain-Specific Modelling Language. In "Proceedings of the 8th Workshop on Multi-Paradigm Modeling (MPM 2014)", CEUR Workshop Proceedings, vol. 1237, p. 21-30, 2014.* Romuald provided the GISMO case study, Bart implemented the approach and applied it to the case study, Romuald, Bart and Tom wrote the paper and Hans reviewed the paper.

16. *Joachim Denil, Bart Meyers, Bart Pussig, Paul De Meulenaere and Hans Vangheluwe. Explicit Semantic Adaptation of Hybrid Formalisms for FMI Co-Simulation. In "Proceedings of the 2015 Symposium on Theory of Modeling and Simulation - DEVS", TMS/DEVS '15, part of the Spring Simulation Multi-Conference, p. 852-859, 2015.* Joachim came up with the idea, Joachim, Bart M. and Bart P. elaborated the idea, Joachim implemented the approach, Joachim wrote the paper, and Paul and Hans reviewed the paper.

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

1.1 Complexity and Modelling languages

The systems we analyse and design today are characterized by an ever increasing complexity. As demands on quality grow, development- and production cost must still be kept minimal. A recent approach to tackle this complexity is Model-Driven Engineering (MDE) [182]. Development of such complex systems spans a plethora of different activities, including requirements modelling, analysis, design, testing, etc. for often very diverse system components and views. This inherent heterogeneity is tackled in *Multi-Paradigm Modelling* (MPM) by explicitly modelling all aspects of the system under study at the most appropriate level(s) of abstraction using the most appropriate modelling language(s) [146, 45]. The goal is to minimize “accidental” (as opposed to “essential”) complexity [25] by capturing only the essence of a problem.

More specifically, in *Domain-Specific Modelling* (DSM) [74] the general goal is to provide means for domain users such as a mechatronics engineers or hospital process managers to model systems using concepts from their problem domain. Techniques such as metamodelling [108] and model transformation [46] enable modelling language engineers and domain experts to create Domain-Specific Modelling Languages (DSMLs) for domain users. DSMLs separate problem domain knowledge from implementation target (software and/or hardware) domain knowledge. As syntax and semantics of DSMLs are precisely defined by means of metamodelling and model transformation, models can be used for analysis, simulation, optimisation, documentation and even full code synthesis. Often, following MPM principles, multiple aspects/views as well as subsystems are modelled using distinct DSMLs. These DSMLs are often specifically designed to address a specific facet of a problem, such as architecture. This implies that not only system modelling, but also modelling language engineering, becomes part of the development process. In this work, *modelling languages* refer to DSMLs rather than *general purpose modelling languages*, such as UML, unless stated otherwise. Language engineering is thus an enabler, if not a prerequisite, for DSM.

The system modelling and language modelling phases are two parallel development processes. In this thesis, we assume that the domain user is responsible for the system modelling phase, and that the language engineer and domain expert are responsible for the language modelling phase. Both development processes may incorporate multiple phases such as analysis, design, implementation (typically automated as code generation), testing, maintenance, etc., which may involve new software engineering roles or stakeholders such as a requirements engineer, quality analyst, usability expert, test engineer, product manager etc. Both phases can follow any design process, such as the waterfall model, incremental development, etc. In [180], the authors testify from industry experience how DSM addresses problems for each of the involved stakeholders, from core developers, to development teams, to top management. However, in this thesis we restrict ourselves to the assumption that all processes at the language level are undertaken by the language engineer and domain expert, all processes at the domain-specific system level are undertaken by the domain user, and that all processes at the platform-specific implementation level are automated (as defined during the language engineering phase). From a user’s point of view, the goal of DSM is that systems can be modelled, simulated, analysed, etc. by non-technical domain users.

Since the development process is extended with a language engineering phase, the question arises under what circumstances it becomes worthwhile to employ DSM. In general, using DSM yields the best results if systems are considerably better expressed in terms of the problem domain rather than general purpose modelling. Moreover, it should be expected that multiple models will have to be created. At Panasonic, Safa [180] concluded that DSM is particularly useful when used over the course of longer projects, in an evolving domain. According to the study, the high customisability and quick development of a DSML enable language engineers to address the specific problems of the domain users. Using state-of-the-art DSM tools result in a tremendous benefit in comparison with using general purpose modelling (*i.e.,* UML, for which there was a mismatch with the available tools, language concepts and evolution support) or assembly language and C programming.

Anecdotal evidence shows five to ten times productivity increase thanks to DSM [96]. Despite some positive reports, DSM still lacks mature tools and techniques hampering widespread industry adoption. In this thesis we address a number of challenges in modelling language engineering, and provide solutions in the form of tools and techniques. The thesis statement is as follows:
This thesis contributes to the engineering of Domain-Specific Modelling Languages (DSMLs) by addressing the challenges of defining verification support for DSMLs, evolution support for DSMLs and composition support for DSMLs.
What follows is a description of the addressed challenges.

The first challenge concerns verification support for DSMLs. In DSM, the general goal is to provide DSMLs for domain users to model systems in their problem domain. Verifying whether a model satisfies a set of requirements is considered an important challenge in DSM [65]. It is nevertheless mostly neglected by current DSM approaches. Verification has been achieved by translating models to formal representations. Logic-based formulas in formalisms such as Linear Temporal Logic (LTL) [162] and Computation Tree Logic (CTL) [58] are used to represent the temporal properties that need to be verified [171]. Currently, domain users need to have a profound knowledge of some logic to express properties. This violates the principles of MPM. As with design models, the level of abstraction for specification and verification tasks needs to be raised to the domain level, as domain users should not be exposed to underlying notations and techniques [210]. In this sense, DSM should not only address modelling the design of a system, but also its properties, its environment, its run-time state, and its execution traces, which should all be modelled at the domain level, in their own DSML.

The second challenge concerns evolution of DSMLs. In software engineering, the evolution of software artifacts is ubiquitous [131]. Diverse artifacts such as requirements, programs, data, tests and documentation may evolve. In MDE, where modelling languages play a central role, evolution occurs not only at the level of models, but also at the level of modelling languages. This is in contrast with general-purpose programming languages such as C++ where programs evolve, but not the programming language (or at least, very infrequently and mostly in a backwards compatible fashion). Language evolution applies in particular to DSMLs, where relatively frequent changes in the problem domain as well as in the implementation target domain (*e.g.,* due to external technical or strategic decisions) must be reflected in the respective languages, as shown in an industrial context by Safa [180]. These changes are necessary to maintain the high coupling between domain and language. A first problem is the need for rapid development techniques for DSMLs, as they are created and modified frequently during the life-cycle of the system they are used for. A second, and far greater problem is that possibly large numbers of modelling artifacts such as instance models or transformation models become invalid and unusable when a related DSML is modified/evolved. Early adopters of MDE and DSM dealt with language evolution issues manually [180]. However, this approach, as well as an ad hoc approach to any language change, is tedious and error-prone [197]. Hence, dealing with evolution requires in-depth knowledge of the language as a whole. Without a proper scientific foundation, as well as methods, techniques and tools to support evolution, MDE in general and DSM in particular, cannot live up to its promise of ten-fold productivity increase [96]. This becomes apparent when projects span longer periods of time [180]. Since the problem of modelling language evolution was first identified by Sprinkle and Karsai [187], the general problem has only grown in importance, yet remained largely unsolved at the time of our research.

The third challenge concerns composition of modelling languages. Current language engineering methods require the language engineer to start from scratch when developing a new DSML. This is in contrast with the observation that there are similarities between DSMLs: one might need a new variant of Petri nets or a different kind of automaton (as there are tens of variants of these two formalisms), or the combination of a number of formalisms. Therefore, there is a clear need for reuse of existing languages, or language modules as illustrated in [159]. This need was emphasised at the 2015 MoDELS conference, where five workshops GEMOC (Globalization of Modeling Languages), EXE (Executable Modeling), ModComp (Model-Driven Engineering for Component-Based Software Systems), FlexMDE (Flexible Model-Driven Engineering) and MPM (Multi-Paradigm Modeling) explicitly addressed the need for language reuse and composition. Moreover, the composition of models in different languages can be very complex, but necessary in MPM. Different parts of a system may belong to different technical domains. Also, different abstraction levels, different aspects of the system, and different phases in a design activity require different modelling techniques and tools. The combination of semantics (in the form of simulation or execution) is especially challenging for combined models with heterogeneous behaviour. It is therefore necessary to deal with the simulation of multi-paradigm, heterogeneous models.

1.2 Background

The necessary background for modelling language engineering is presented in this section.

1.2.1 Modelling Language Engineering

The basis for this thesis can be found in Domain-Specific Modelling (DSM) [96]. DSM is part of model-driven engineering (MDE). It requires modelling systems using domain concepts, rather than concepts in the solution domain (*i.e.,* the computing domain). Consequently, systems are modelled at a higher level of abstraction, and often code is generated from these high-level models. This means that the development process is split into two tasks: (i) the creation of a Domain-Specific Modelling Language (DSML) known as *engineering a language*, by the *language engineer*, in consultation with a domain expert, and (ii) modelling the system using this DSML by a problem domain (but not solution domain) user, referred to as a *domain user*. Once the DSML is created, any system in the domain can be modelled in the DSML. Since the challenges addressed in this thesis are an addition to language engineering, this section will mainly discuss the language engineering phase (as opposed to the system modelling phase) of the development process. The topic of modelling languages is developed in the modelling community [73].

The three main aspects of a DSML, or modelling language in general, are its *abstract syntax* (describing the internal structure of a model, as *e.g.,* a typed abstract syntax graph), the *concrete syntax* (describing how a model is represented, *e.g.,* in 2D vector graphics or in textual form) and its *semantics* (describing what a model means, by *e.g.,* a definition of its execution semantics, or a translation to a known formalism) [146].

In this thesis, we assume that the abstract syntax is described by a metamodel [108] in the form of a class diagram [6] or a similar formalism, including additional static semantics specified as constraints. Static semantics can be described in a constraint language such as the Object Constraint Language (OCL) [152]). Instance models of the DSML are said to *conform to* or *typed by* the metamodel. In [108] this relation is referred to as a *linguistic instance of*, and throughout this thesis, *instance (of)* will refer to this kind of instantiation. Conformance usually means that there is a morphism between the instance model and the metamodel. Both can always be represented as typed, attributed, directed graphs [108], which assumes a discrete (as opposed to continuous) representation. Models that include continuity can be represented intensionally by a graph as well, by for example as a model of the structure of the associated mathematical formula, as often done in Simulink® [128]. In particular, programming code can be represented as a tree (*i.e.,* an abstract syntax tree), which is a kind of graph.

The abstract syntax is represented by at least one concrete syntax, either textually or graphically (or a combination of both). Mappings between the abstract syntax and its concrete syntax(es) are called *rendering functions*, and their inverses are called *parsing functions*. These functions are used by model editors to render (changes of) the model's abstract syntax, and to parse user's concrete syntax edits to abstract syntax. In case of graphical concrete syntax, we use a connection-based syntax [37] in this thesis. This means that concrete syntax models consist of icons (one kind for each metamodel class) that are connected by splines (for each metamodel association). The other class of visual languages proposed by Costagliola *et al.* [37] is geometric-based syntaxes, where objects are spatially arranged. The authors consider a textual, string-based representation a special kind of geometric-based syntax.

Semantics can be described translationally or operationally. Translational semantics provide a mapping of a model in a given modelling language onto a model in a different modelling language for which a semantics is available. Operational semantics capture explicitly how a model can be executed, also called *simulating* the model, which is effectively mapping a model onto a trace. In this thesis, semantics are always formalised as transformations, that can transform (an) input model(s) to (an) output model(s), instances of the same or different languages. Since models are represented as typed, attributed, directed graphs, a popular way to specify transformations is by means of graph transformation rules.

1.2.2 Transformation Models

Transformation models can be classified according to their syntax [133]. Endogenous transformations use the same metamodel for their input and output models, while exogenous transformations use different metamodels. Unidirectional transformations are transformations from one model to another, in that specified direction. Bidirectional transformations specify a relation between models, which can be operationalised in both directions. Out-place transformations or model-to-model transformations create a new output model, while in-place transformations perform changes on the input model itself.

Transformations can be explicitly specified as transformation models [17], in a language that combines generic transformation concepts such as rules and a rule schedule, and concepts specific to the languages it transforms.

A *rule schedule* of a transformation language is generic, and allows the modelling of how different rules are scheduled. *Rules* consist of a *left-hand side* (LHS) containing a pattern representing a condition, and a *right-hand side* (RHS) containing a pattern representing an action (elements can be created, removed or updated). LHS and RHS are generic language constructs that can contain elements, each displayed as a different shape of container. The contained elements form graph patterns that reuse concrete syntax taken from the input and output language. When a rule is evaluated, a match for the LHS is searched for in the input model. If a match is found, the RHS is *applied* to the input model, thus changing it. If the rule *fails* to match, the input model is left untouched. Depending of the outcome (failure or application), the next rule according to the rule schedule is evaluated. Rule schedules may be implicit. For example, when executing the transformation, a random matching rule may be applied.

A process called RAMification to generate such a rule-based domain-specific transformation language for given input and output DSMLs is presented by Kühne et al. [110]. The name RAMification for the process of generating a domain-specific transformation language is an abbreviation of its three stages to generate a pattern language (for LHS) and action language (for RHS) from the DSML:

1. *Relaxation*: the metamodel is relaxed by removing constraints in order to allow the creation of partial models, to uniquely identify elements across the different parts (LHS/RHS) of a rule;

2. *Augmentation*: each element in the metamodel is augmented with a label, and with an option whether subtypes should be potential match candidates. Other augmentations can be used to influence the matching process [110] but are not relevant for this thesis;

3. *Modification*: for (attributes of) elements of the LHS, conditions over values are specified because the LHS pattern essentially models a condition or constraint that must be met. Similarly, for elements of the RHS, assignments of values are specified.

Since the result of the RAMification transformation is a language definition with metamodel and concrete syntax model, the language engineer can adapt the concrete syntax model to make rules more appealing. The three stages of the RAMification process are explained more in detail using the running example in Section 1.3.

The RAMification process is implemented as a transformation that takes a DSML metamodel and concrete syntax model as input and a metamodel and concrete syntax model of the transformation language as output. This is called the RAMification transformation [110]. Since transformations themselves are explicitly modelled, and written in a modelling language, transformation models conform themselves to a transformation language, that has abstract syntax and concrete syntax. Consequently, transformation models can be transformed in their own right, or can be generated. The transformations that have transformation models as input and/or output are called higher order transformations (HOTs). Similarly, abstract syntax and concrete syntax of a

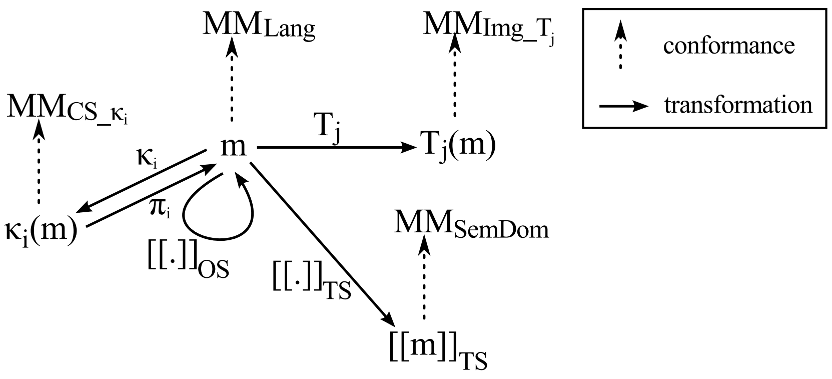


Figure 1.1: A model and its relations in the context of MDE.

language are modelled as metamodels (in the class diagram language) and concrete syntax models (in *e.g.,* an icon language) respectively, and can thus be transformed as well.

1.2.3 A DSML Relational Ecosystem

Figure 1.1 shows the different kinds of relations involving a DSML. We call this the *relational ecosystem* of the DSML. A relational ecosystem can be studied as a whole, *e.g.,* to investigate evolution of artifacts. Relations are visualised by arrows, "conforms to"-relationships are dotted arrows. The instance model *m* conforms to a metamodel MM_{Lang}, the abstract syntax model of the DSML *Lang*. There is a rendering function κ_i between *m* and a concrete syntax $\kappa_i(m)$ model. The inverse of κ_i is a parsing function π_i so that $\pi_i(\kappa_i(m)) = m$. The index i highlights the fact that multiple concrete representations may be used. $\kappa_i(m)$ conforms to a metamodel $MM_{CS\_\kappa_i}$, the explicit model of the concrete syntax language (such as the set of all 2D vector graphics drawings). Semantics can be described transformationally or operationally. Both are included in Figure 1.1. Transformational semantics are described by the semantic mapping $[[.]]_{TS}$, and maps *m* to a model $[[m]]_{TS}$ in the semantic domain. This semantic domain is a different modelling language with its own syntax en semantics. Similar to *m* conforming to MM_{Lang}, $[[m]]_{TS}$ conforms to MM_{SemDom}. Operational semantics are described by the semantic mapping $[[.]]_{OS}$. It is an in-place, endogenous transformation that iteratively makes changes to a model to thus producing an execution behaviour. Additionally, transformations T_j may be defined for *m*. The index j highlights the fact that multiple general transformations (*e.g.,* for synthesis, migration, abstraction/refinement, normalisation, optimisation, etc. [133]) may exist.

Assuming that a modelled system consists of models in an appropriate language and explicit relations between these models, we posit that the relational ecosystem of any modelled system can be mapped on (multiple instances of) Figure 1.1. Multiple models *m* can exist in a system, typically all conforming to the same metamodel MM_{Lang}. Each model can be mapped on Figure 1.1 separately. Note that $T_j(m)$ or $[[m]]_{TS}$ can themselves be seen as a model *m* in Figure 1.1, to which possibly relations such as *T* and $[[.]]_{TS}$ apply. Since languages are explicitly modelled the metamodel MM_{Lang} of a DSML is an artifact that is part of the modelled system in its own right. Thus, MM_{Lang} itself can be seen as an *m* in Figure 1.1, having a metamodel in its own right. Note that programs can also be considered as models, with an abstract syntax tree, a concrete syntax and a semantic mapping to, for instance, machine code or an operational semantics in the form of an interpreter.

In conclusion, Figure 1.1 describes any explicitly modelled system. Although the statement that all modelled systems can be mapped to the above diagram cannot be formally proved, we are confident that it holds, based on our experience with DSM. We assume Figure 1.1 describes any explicitly modelled relational ecosystem, thus allowing us to identify or relate challenges in this thesis.

1.2.4 Formalism Transformation Graph and Process Model (FTG+PM)

It has become clear that the DSM process involves different models (*e.g.,* domain-specific models, metamodels, transformation models, etc.) in various modelling formalisms (DSMLs, class diagrams, transformation languages, etc.) that each serve a specific purpose in the DSM process. Indeed, in accordance with MPM principles multiple modelling languages may be used to tackle complex problems. MPM approaches use multiple modelling languages and deal with their consistency and interaction. This means that the MDE process for development and/or execution of the modelled system becomes more complex as well, and is highly customised.

The Formalism Transformation Graph and Process Model (FTG+PM) [125] captures the MDE process by explicitly modelling it. The FTG+PM consists of the Formalism Transformation Graph (FTG) and its complement, the Process Model (PM), and models all kinds of activities in the MDE lifecycle such as requirements development, domain-specific design, verification, simulation, analysis, calibration, deployment, code generation, and execution. The FTG describes in an explicit and precise way formalisms, and their relationships, as transformations between models in these formalisms. The PM specifies an MDE process using these formalisms and transformations, and can be used as a basis for process enactment. Figure 1.2 shows a FTG+PM of the DSM process.

The FTG depicts (domain-specific) formalisms as labelled rectangles. Transformations (in the broad sense of the word) between models in those formalisms are depicted as labelled small circles. Formalisms can be connected to transformations by arrows that depict formalisms as in- and output of these transformations: executing a transformation takes instances of incoming formalisms as input, and creates instances of outgoing formalisms as output.

The PM is specified in the UML Activity Diagram 2.0 language [153]. The ovals (actions) in the Activity Diagram correspond to executions of the transformations declared in the FTG. Labelled rectangles (data objects) in the PM correspond to models that are consumed or produced by actions, and are instances of the formalisms in the FTG with the same label. Thin arrows in the PM indicate data flow, while thick arrows indicate control flow. Similar to the models, the arrows must also have corresponding arrows in the FTG, meaning that their input and output nodes must correspond. We also use Activity Diagrams control flow constructs for a PM such as joins and forks, represented by horizontal bars, and decisions, represented by diamonds. (Executions of) transformations can be automatic (shaded) or manual (white). The meaning of the FTG+PM is presented in depth in [125].

We use the FTG+PM in the context of DSM, to describe both the language engineering phase and the system modelling phase, while the FTG+PM was up to now only used for the latter [149]. Since languages are created in the language engineering phase and are later used in the system modelling phase, they need to appear on both the PM and FTG side. To this end, we extend the

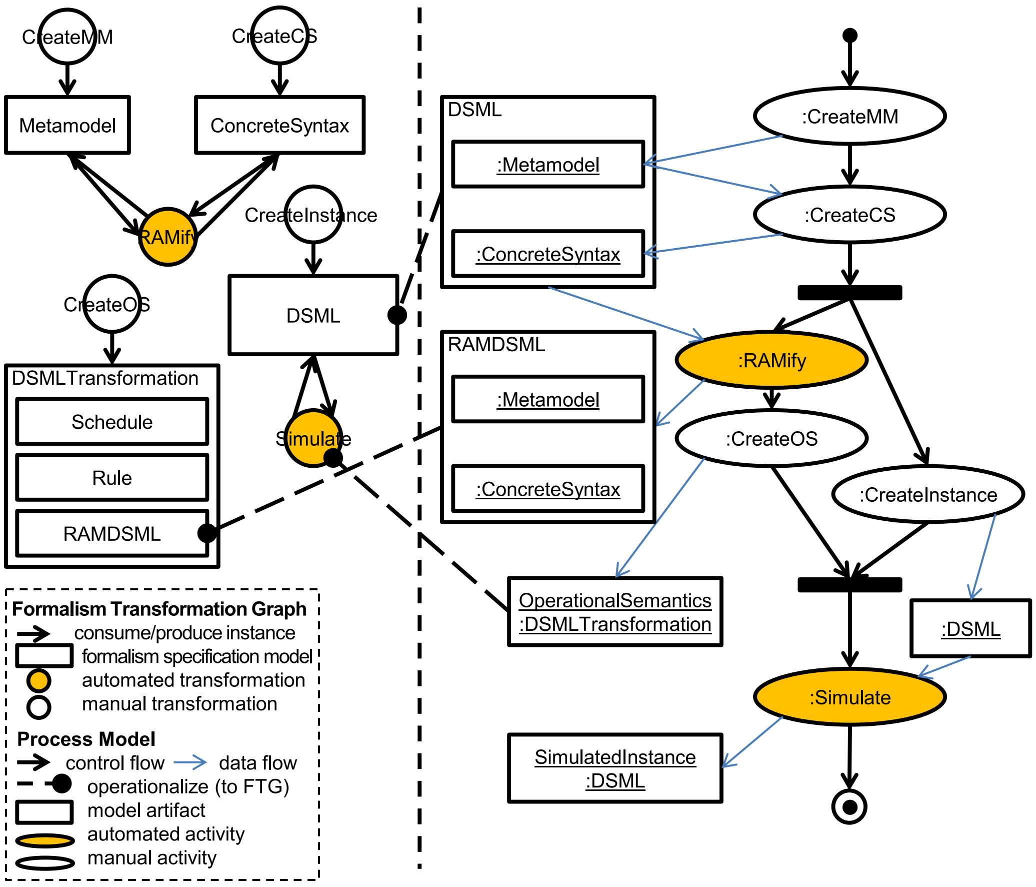


Figure 1.2: A FTG (left) and PM (right) of the DSM process.

FTG+PM language with the operationalise relationship. Using the operationalise relationship, a metamodel and concrete syntax model can be *put into operation* as a language, meaning models that conform to the language can be created. A tool can automate this as compilation or interpretation of the metamodel and concrete syntax model, possibly by generating a (syntax-directed) language-specific editing environment. Note the box around metamodel and concrete syntax model to denote that they are operationalised together to form a language (semantics is not yet included in the box). Similarly, a transformation model can be put into operation by compiling or interpreting it, meaning it becomes executable, often to give semantics to a language. In the process model of Figure 1.2 all activities from the top up to the CreateOS activity are performed by the language engineer, and the remaining two activities are performed by the domain user.

More in detail, the language engineer first creates a metamodel in the CreateMM activity. Then he creates a concrete syntax model in the CreateCS activity as well as a mapping specifying the bidirectional relationship between both (note that this is not visible in Figure 1.2), which results in a DSML definition. Then, he/she RAMifies the DSML in the automatic RAMify transformation. The resulting RAMDSML is part of the DSMLTransformation language. This is the language of the OperationalSemantics transformation model, which is subsequently created in the CreateOS activity. This concludes the creation of a DSML. Next, the domain user models a system in the CreateInstance activity. He/she can simulate this model, by executing the operational semantics in the Simulate activity, resulting in a new DSML instance.

Note that relationships between artifacts might become visible when following links transitively. For example, in the PM, one might expect that in the PM the OperationalSemantics model should be input to the Simulate activity. The OperationalSemantics model is however not literally an input of Simulate. Instead, the Simulate transformation in the FTG is an operationalisation of the OperationalSemantics model. Thus, the Simulate activity in the PM is implicitly related to the OperationalSemantics model.

1.2.5 Tools for Domain-Specific Modelling

The following tools for domain-specific modelling are used in this thesis.

AToM^3

AToM^3 stands for *A Tool For Multi-Formalism and Metamodelling* and supports syntax-directed modelling [46]. In *AToM^3*, abstract syntax, concrete syntax and semantics of a language can be modelled explicitly. The tool supports metamodelling and model transformation, and supports the generation of domain-specific modelling environments. Models have graphical connection-based concrete syntax [37]. Geometric-based concrete syntax can be emulated by using action code that is triggered by various events (*e.g.,* creation, movement, etc. of an element, or saving the model). However, this action code, in the form of Python code, is not explicitly modelled. Since models are represented as graphs, model transformation is performed through graph transformation, and rules inherit the concrete syntax of the DSML(s) that are to be transformed. However, RAMification is not implemented in this tool, so HOTs are not directly supported. The transformation of metamodels is in practice not supported, as parts of the metamodelling language is not explicitly modelled. A detailed explanation of *AToM^3* be found in [47].

AToMPM

The successor of *AToM^3* is *AToMPM*. *AToMPM* stands for *A Tool for Multi-Paradigm Modelling* and is presented in [126]. *AToMPM* improves on *AToM^3* by supporting a complex rule scheduling language, RAMification, multiple concrete syntaxes for a single abstract syntax and collaborative modelling. Still under development, it aims to tackle many other challenges found in other DSM tools by explicitly modelling an action language, textual syntax, model management, debugging, tool behaviour and user interaction. Additionally, the tool has a Web-based user interface.

metaDepth

metaDepth [42] is a tool for textual modelling language engineering and modelling that allows the definition of the three aspects of a language: abstract syntax, concrete syntax and semantics. Abstract syntax is modelled using the *metaDepth* default textual syntax, which can be used on any metalevel. Domain-specific textual concrete syntax can be optionally modelled using a designated language, which enables assigning the metamodel and each metaclass a syntactic template indicating how their instances are represented. Models conforming to *TextualSyntax* are compiled to ANTLR grammars[1], with appropriate semantic actions to create the model in memory [43]. *metaDepth* integrates the Epsilon family of model management languages[2]. Hence, behavioural semantics are defined with an in-place transformation using the Epsilon Object Language (EOL) [105], a language similar to OCL extended with imperative constructs to create objects and variables, and assign values. Moreover, Epsilon includes the Epsilon Generation language (EGL) for template-based code generation. Epsilon includes a number of other languages, each with their own purpose including a rule-based transformation language ETL. Only EOL and EGL are used in this thesis. The use of Epsilon means that HOTs are not supported in *metaDepth*. *metaDepth* allows deep metamodelling [42], to abandon the rigid metamodel/model architecture, by using potency. We do not make use of this feature in this thesis. Nevertheless, top level metamodels cannot be easily transformed since the metamodelling language is not explicitly modelled.

Discussion

Each of these three tools can be used for DSM, but each has its own advantages and drawbacks. Many of the (dis)advantages stem from the usability of the tools. *AToMPM*, still under development, currently suffers from low usability. Nevertheless, *AToMPM* is designed to model visual concrete syntax explicitly, including action code. In the current version, the latter is unfortunately not yet implemented. According to MPM principles, metamodels and transformations are modelled explicitly and are conform to a modelled language, thus allowing metamodel transformations and higher order transformations. *AToM^3* lacks this language manipulation support, but is otherwise easier to use, and models are easier to manipulate and debug. Its transformation language is notably less expressive, as it lacks explicit rule scheduling. *AToM^3* is used if an advanced concrete syntax is desired (heterogeneous modelling) as geometric-based concrete syntax can be emulated easily. *AToMPM* and *AToM^3* are very comparable otherwise. *metaDepth* is textual, and thus allows for quick prototyping and debugging, as it does not require reopening and editing dialog boxes, reloading transformations, etc. Additionally, the support of Epsilon allows advanced model manipulation, in only a few lines of code.

As a conclusion, throughout this thesis, *AToM^3* is used in case visual concrete syntax is important, *AToMPM* is used in case of advanced language manipulation is required, and *metaDepth* is used for textual modelling (although this a side-effect rather than a goal in this thesis), quick prototyping, and code generation.

1.3 Running Example

The running example that is used throughout this thesis is a DSML for elevator controllers. Our DSML enables modelling a building with floors, elevators and buttons. Additionally, it defines the step-wise (discrete-time) behaviour of an elevator system, such as moving up or down to a different floor, closing or opening elevator doors, or pressing buttons. The example is inspired by an elevator controller model used to illustrate model checking in [134]. A similar "Elevator Control System" is presented Strobl and Wisspeintner [191] where it is used to demonstrate AutoFocus, a tool for

[1]`http://www.antlr.org/`
[2]`http://www.eclipse.org/epsilon/`

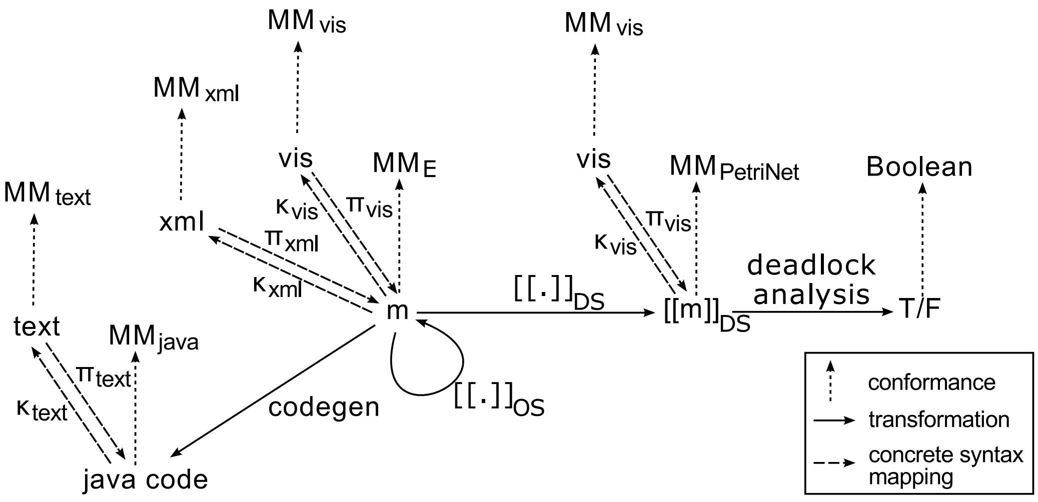


Figure 1.3: Relational ecosystem of the Elevator DSML.

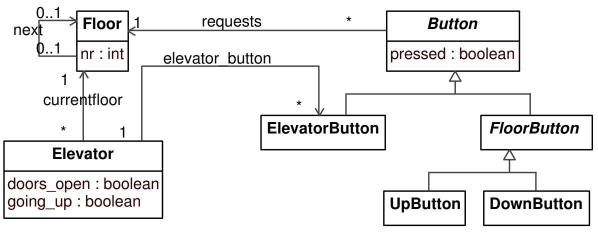
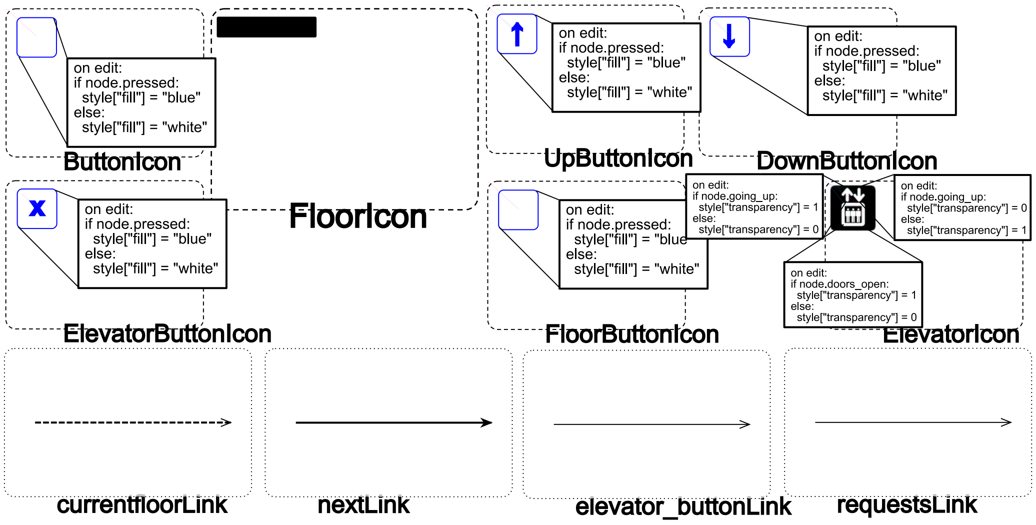


Figure 1.4: The metamodel of the Elevator controller DSML that serves as the running example throughout this thesis.

modelling embedded systems [90]. Although embedded systems modelling is not the focus of this thesis, the work illustrates the complexity of elevator controllers.

The *Elevator* DSML will also be called the *Elevator* language, *Elevator* or the abbreviated E (often in Figures) throughout this thesis. *Elevator* and its relations are shown in Figure 1.3. Its abstract syntax MM_E is defined by means of a metamodel. Its concrete syntax MM_{vis} is defined by means of an icon language. Its semantics are defined in two ways: operational semantics $[[.]]_{OS}$, and transformational semantics $[[.]]_{PN}$ by mapping to the Petri net formalism. The diagram represents the *Elevator* relational ecosystem. Multiple concrete syntaxes MM_{vis} and MM_{xml} (a XML representation) can be used. From instance models e, Java code can be generated. Deadlock analysis can be carried out on the Petri net representation of e, yielding a result in the Boolean domain: *true* or *false*. Concrete syntax for Java and Petri nets is explicitly included in the diagram, because all languages, including programming languages, can be considered modelling languages, if abstract syntax, concrete syntax and semantics are explicitly modelled. In the remainder of this section, a number of artifacts from Figure 1.3 are presented, as they will be used throughout the thesis.

Figure 1.4 shows the metamodel of the *Elevator* language. Elevators move between Floors.



```
on edit:
if node.pressed:
  style["fill"] = "blue"
else:
  style["fill"] = "white"
```

```
on edit:
if node.pressed:
  style["fill"] = "blue"
else:
  style["fill"] = "white"
```

```
on edit:
if node.pressed:
  style["fill"] = "blue"
else:
  style["fill"] = "white"
```

```
on edit:
if node.pressed:
  style["fill"] = "blue"
else:
  style["fill"] = "white"
```

```
on edit:
if node.pressed:
  style["fill"] = "blue
else:
  style["fill"] = "white"
```

```
on edit:
if node.going_up:
  style["transparency"] = 1
else:
  style["transparency"] = 0
```

```
on edit:
if node.going_up:
  style["transparency"] = 0
else:
  style["transparency"] = 1
```

```
on edit:
if node.doors_open:
  style["transparency"] = 1
else:
  style["transparency"] = 0
```

Figure 1.5: The concrete syntax model of the Elevator controller DSML that serves as the running example throughout this thesis.

responding to Button press requests. A Button requests exactly one Floor.Floors are ordered by the next association and a derived attribute nr representing the Floor number. At any time, an Elevator is at exactly one Floor, modelled by the currentfloor association. An ElevatorButton is a button inside an Elevator, allowing a passenger to request going to a certain Floor. At every Floor, there can be an UpButton to request to go up and a DownButton to request to go down. An Elevator can have its doors open (in that case it cannot move) and has a direction (up or down).

A concrete syntax model for the Elevator language in the form of icons and arrows is shown in Figure 1.5. For every element of E, there is exactly one associated Icon or Link. An icon/arrow inside a dashed box with a class or association name beneath it associates that concrete visual syntax with the corresponding abstract syntax. An icon consists of graphical elements, and text elements. This model defines the appearance of instance elements and how these appearances can change (*i.e.,* text content of text elements, or colours or transparency of graphical elements) in concert with their associated abstract syntax, by implementing specific rendering and parsing functions. In this way, as shown in the code fragment connected to all ButtonIcons, a button instance icon has a white fill which will turn light blue if the associated button instance's attribute value for pressed is *true*, *i.e.,* when the button is lit. If the elevator is going up (going_up is *true*), then only the upward arrow is visible. If going_up is *false*, only the downward arrow is visible. Whether or not the doors are open is also visualised.

Figure 1.6 shows an instance model with three floors, one elevator and seven buttons, that uses the concrete syntax. As defined in the concrete syntax model, pressed buttons have a blue fill, and they are connected to the floor they request. On the middle floor, a button is pressed by someone requesting to go down, and inside the elevator the button to go to the top floor has been pressed. The elevator is currently at the bottom floor. Its doors are open and its current direction is down. Note how all elements, including the links, conform to the metamodel of Figure 1.4. For simplicity, we limit ourselves to models with only one elevator. A system with multiple elevators

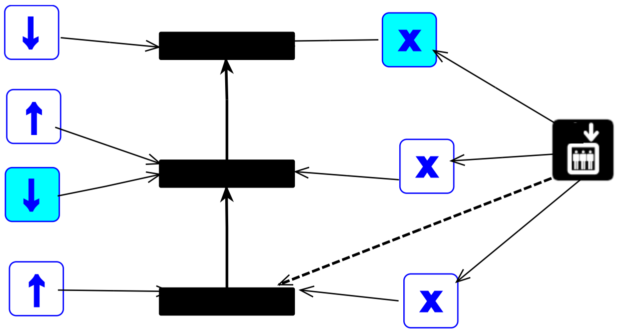


Figure 1.6: An Elevator model, instance of the Elevator DSML.

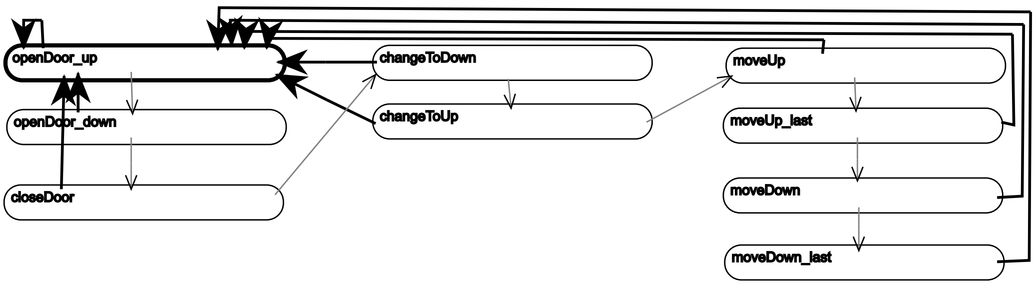


Figure 1.7: Rule schedule of the operational semantics of the Elevator example.

is considerably more complex and would not contribute significantly to explaining the approaches presented in this thesis.

Figure 1.7 shows the transformation model $[[.]]_{OS}$ specifying the operational semantics, that uses the instance model as input. This model is a *rule schedule*, and determines how different rules are scheduled. *Rules* consist of a *left-hand side* (LHS) containing a model pattern, and a *right-hand side* (RHS) containing an action. When a rule is evaluated, a match for the LHS pattern is searched for in the input model. If a match is found, the RHS is applied to the input model, thus changing it in-place. In the case that the rule was *applied*, and an outgoing *success* link in the schedule (depicted as a black arrow) is followed. If no match is found, the input model is unchanged and an outgoing *notApplicable* link (depicted as a grey arrow) is followed. Execution starts at openDoor_up, since it has the isStart flag set (depicted as a bold line of the rule).

Inspired by a real elevator controller, the following rules implement how the elevator changes floors (one at a time), and opens and closes its door to honour the requests of users (modelled as pressed buttons). Three rules implementing this behaviour are shown in Figure 1.8. When a request for a floor is made for a different floor than the elevator's current floor, the doors close so that the elevator can start moving. This is modelled in the closeDoor rule shown in Figure 1.8. The LHS shows the pattern denoting an elevator with open door on a floor. Its direction is greyed out as it is not relevant to match this pattern, *i.e.*, it will match any direction. The pattern also shows a lit button (any subtype of Button, hence the icon of the abstract Button class is shown) on another floor, corresponding to the condition described above. The number labels on the top left of each pattern element serve as the relationship between LHS and RHS. Elements in the RHS with the

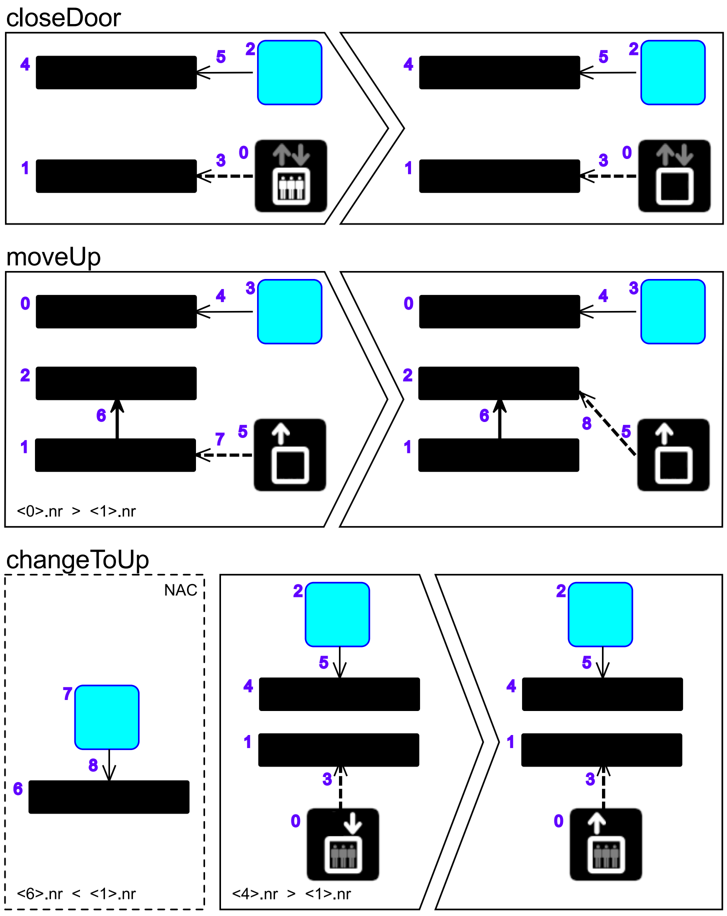


Figure 1.8: Three rules taken from the operational semantics of the Elevator example.

same label, are the same elements. The action in the RHS here denotes that none of the matched elements are changed, except for the doors_open attribute value of the elevator. Further rules in the transformation denote that the elevator passes all floors that are requested on its path (which is either up or down), and opens its door when the elevator's direction corresponds to the direction requested on that floor. The case where the elevator is moving up (*i.e.,* changes its currentfloor link) is shown in the moveUp rule. Note that the LHS contains not only a pattern, but also a constraint, stating that the requested floor (with label 0) must have a higher floor number than the current floor (with label 1). Related rules (not depicted) are moveDown (the dual of moveUp), moveUp_last (where the lit button is on the next floor), and moveDown_last (the dual of moveUp_last). Pressed buttons unlight when the door opens at a requested floor and the elevator goes in that direction in the openDoor_up rule (in the case the elevator is going up) and its dual, the openDoor_down rule. The elevator only changes its direction if there are no more requests on its path. This can be seen in the changeToUp rule, which contains a negative application condition (NAC) (visualised as a dashed box) in addition to its LHS. A NAC is evaluated the same way as a LHS, but the rule will *not* be applied if a match is found. Hence, the rule will be applied if a button is pressed at a higher floor (modelled in the LHS), but not if a button on a lower floor is lit at the same time (modelled in the NAC), *i.e.,* in its current path. Note that, if the elevator is at a lower floor and is going up, it can pass by a floor where one has requested to go down without stopping, as the elevator is going in the opposite direction, and vice versa.

The first rule that is applicable to the instance model of Figure 1.6 is the changeToUp rule. The LHS Elevator pattern element is bound to the only elevator in the instance model, and the currentfloor with label 3 and the Floor with label 1 are bound to the currentfloor link to the bottom Floor. Additionally, pattern elements with label 2, 4 and 5 can be bound to the middle Floor and lit DownButton with request link in between. An alternative match for pattern elements with label 2, 4 and 5 can be found as the top Floor and lit ElevatorButton with request link in between. No match for the NAC can be found since the elevator is at the bottom floor and going down, thus no buttons are in the elevator's path. Consequently, the rule matches and the RHS can be applied, in which case only pattern element 0 that is bound to the node in the instance graph changes an attribute value (*i.e.,* in the elevator, going_up is set to *true*).

If no rule is applicable, the transformation terminates. This means that the operational semantics implement the behaviour in response to the initially pressed buttons. Unlike in a realistic scenario, buttons can not be pressed during execution. One could incorporate this behaviour in $[[.]]_{OS}$, by adding a rule in which a button is pressed. This however mixes input to the system, with its reactive behaviour, and can therefore be considered bad practice. Rather, this should be modelled separately in an input language.

Note that if the rules are scheduled differently, *i.e.,* if the move rules are scheduled before the change direction rules, the NACs in the change direction become superfluous. We use this $[[.]]_{OS}$ in the running example, to be able to showcase support for NACs where necessary.

As can be seen from Figure 1.8 the modelling language for rules is composed from some generic language constructs for LHS, RHS and NAC, each displayed as a different shape of container. They include a constraint or action, and domain-specific language constructs that borrow syntax from the DSML. As stated before, this language can be generated using a transformation that takes the DSML metamodel and concrete syntax model as input and the metamodel and concrete syntax model of the transformation language as output, called the RAMification transformation [110]:

1. *Relaxation*: the metamodel is relaxed in order to allow for patterns which are partial models: constraints on lower multiplicities in the metamodel are removed, abstract classes are made concrete so they can be instantiated (see also the buttons used in rules of Figure 1.8), and global constraints are removed. For example, Buttons and FloorButtons become instantiable, and a Button does not need to be connected to a Floor in a pattern (but does in a model conform to the *Elevator* metamodel);

2. *Augmentation*: each class and association in the metamodel is augmented with a label attribute for pattern binding (its concrete syntax is a label in the top left corner of the icon or next to the link), and each class with subclasses is augmented with a Boolean attribute denoting whether subtypes should be matched. Other augmentations can be used to influence the matching process [110] but are not relevant in the context of this thesis;

3. *Modification*: for elements of the LHS, the type of each attribute is changed to the type Condition because the LHS pattern essentially models a condition or constraint that must be

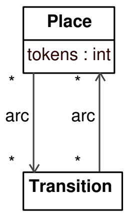


Figure 1.9: A metamodel for the Petri net formalism.

met. For example, a lit button in a LHS has pressed == True as value of its pressed attribute. Similarly for elements of the RHS the type is changed to Action, which allows assigning new values to attributes. For example, a lit button can be turned off in the LHS by setting pressed = False as value of its pressed attribute.

Since the result of the RAMification transformation is a language definition with metamodel and concrete syntax model, the language engineer can adapt the concrete syntax model to make rules more appealing. An example of this is shading the elevator direction or doors grey in the icon to denote that there is no constraint on the respective attribute values.

A second transformation $[[.]]_{PN}$ is defined to represent the transformational semantics to Petri nets [160]. The Petri net formalism is a simple language for modelling non-deterministic discrete distributed systems. Because of its simplicity, it can be subjected to a wide range of analysis techniques, such as verifying whether the system can deadlock (as shown in Figure 1.3). Therefore, it is often useful to map a DSML to a well-supported language such as Petri nets.

A metamodel for the Petri net formalism is shown in Figure 1.9. It consists of Places that can be connected by arcs to Transitions and vice versa. Places can hold a number of tokens. The semantics of Petri nets can be easily described informally. A Petri net structure is a collection of Places and Transitions connected with arcs. There is an initial configuration, with a specified number of tokens in each Place. A Transition is said to be enabled if all of its incoming Places (*i.e.,* Places with an arc to the Transition) have a tokens value of at least 1. The Petri net is simulated by firing a randomly chosen enabled Transition. This means that the tokens value of all incoming Places is decreased by 1, and the tokens value of all outgoing Places is increased by 1. This results in a new configuration, after which again a randomly chosen enabled Transition is fired, and so on.

The transformation $[[.]]_{PN}$ is an exogenous out-place transformation, creating a new model conform to the Petri net formalism, from an *Elevator* model. Like $[[.]]_{OS}$, this transformation assumes the presence of only one Elevator. A selection of semiformal rules of the transformation is shown in Figure 1.10. The schedule is not shown, but specifies that all rules are executed once for every match of the LHS. The following rules are shown:

— the system rule has an empty LHS, and it is only executed once. It generates a fairness loop between system (the reactive behaviour of the elevator controller) and environment (the user pressing buttons);

— the button rule transforms Buttons to two Places representing the Button being off (not pressed) or on (pressed). A place-antiplace pattern is used: at any time, exactly one of these Places has one token. A transition from on to off is added by a different rule (not shown) because it

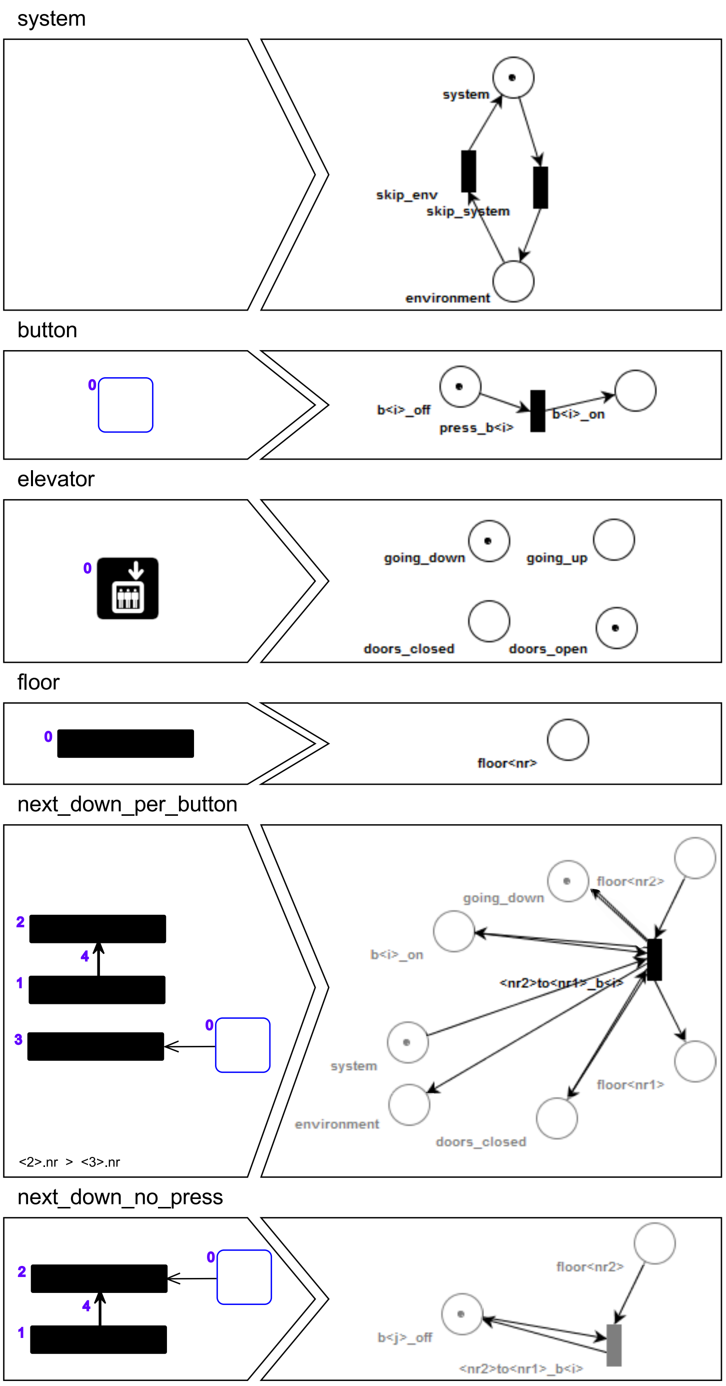


Figure 1.10: The $[[.]]_{PN}$ transformation (partial).

involves other places as well. Place and Transition names are given corresponding to the Button id, thus establishing an name-based link between the Button and its Petri net counterpart;

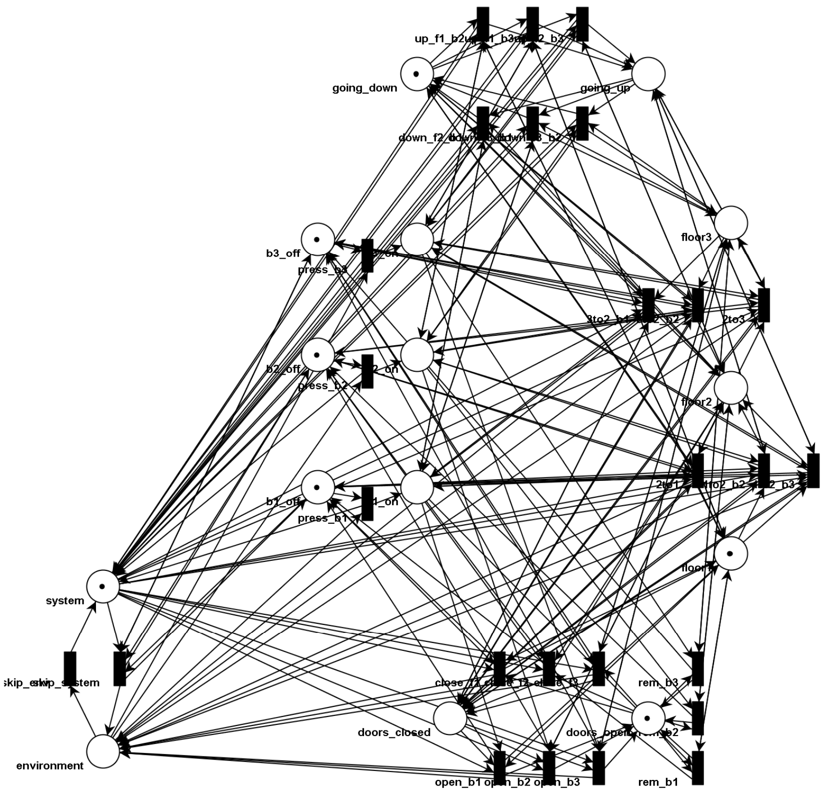


Figure 1.11: The generated Petri net of the *Elevator* instance model.

— the elevator rule transforms an Elevator to two place-antiplace patterns: one for the going_up attribute, and one for the doors_open attributes. Again, the transitions between the places and antiplaces are generated by another rule;

— the floor rule transforms each Floor to a place in the floor rule. Its tokens value is 1 if the Elevator is on that Floor;

— the next_down_per_button rule implements that an elevator can go down if the Elevator's going_up and doors_open attributes are both *false*, and a Button is pressed on a lower Floor. This semiformal rule builds on previous rules. It assumes that all greyed-out Petri net elements are available (*i.e.,* they are part of the LHS as well), and they are matched by name according to their associated Elevator counterparts. The rule creates a Transition and arcs for every Button that is on a lower Floor;

— the next_down_no_press rule implements an extra condition for changing floors: no Button can be pressed that requests the Floor that is left.

The result of transforming the *Elevator* instance of Figure 1.6 to Petri nets is shown in Figure 1.11. This model now supports analysis. For instance, if deadlock analysis is performed in *Pipe2*, a tool for Petri net modelling and analysis [21], the tool determines that no deadlock can be found. However, it is clear that the resulting Petri net is not as easily understandable as the domain-specific model, thus illustrating the advantage of using DSM.

All models shown in this section are *AToMPM* models, except for metamodels (which are modelled in the UML tool MagicDraw as they are too large for visually nice representation in *AToMPM*) and Petri net instances (which are modelled in the Petri net tool *Pipe2* as this tool supports analysis of Petri nets – the necessary code generator to *Pipe2*'s format PNML is not included in Figure 1.3, for simplicity). None of the other modelling artifacts displayed in Figure 1.3 are further explained, as they are not explicitly used in this thesis. More languages and models, with their own specific purposes, can be introduced in the *Elevator* relational ecosystem according to the user's needs and preferences by establishing their relationship with E. Figure 1.3 represents one example of the *Elevator* DSML in relationship to other modelling artifacts.

Throughout the thesis, the adjective *traditional* will mean *according to the state of the art as described in Chapter 1.*

1.4 Research Questions, Contributions and Assumptions

After presenting the necessary background, research questions can be asked that follow from the problem description in Section 1.1. Three challenges in modelling language engineering were identified: verification for modelling languages, evolution of modelling languages, and composition of evolution languages. We presented the following thesis statement:
This thesis contributes to the engineering of Domain-Specific Modelling Languages (DSMLs) by addressing the challenges of defining verification support for DSMLs, evolution support for DSMLs and composition support for DSMLs.

We narrow down these challenges to the context of DSM as presented in Section 1.2. We assume that modelling languages are defined explicitly as described above, by a metamodel, a concrete syntax model (either textual or visual) and semantics (operational or translational) as a transformation. These executable transformations have discrete-time semantics.

The challenge of verification for modelling languages leads to the following research question:
RQ 1. Can we find a general MPM solution to support the specification of (temporal) properties, as well as their verification? This solution should include support for verifying these properties. The user should only be exposed to domain-specific concepts. The solution should weigh as little as possible on the lightweight DSM process, and should be an extension of the existing DSM techniques for design languages.

The challenge of evolution of modelling languages leads to the following research question:
RQ 2. What is a solution according to MPM principles for dealing with the consequences of language evolution?

To address the challenge of composition of modelling languages, we selected two complimentary research threads. The third research question takes a syntactic approach:
RQ 3. Can we extend generic model composition mechanisms, inspired by generic programming, to the composition of DSMLs, in its three aspects, namely, abstract syntax, (textual) concrete syntax and (operational) semantics?
The fourth research question concentrates on the semantics of modelling languages:
RQ 4. What is a solution according to MPM principles for dealing with execution of heterogeneous modelling and semantic adaptation?

The contributions of this thesis are the answers to these research questions. The goal for all contributions is to adhere to MPM principles: everything should be modelled explicitly, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s). As users should only be exposed to such formalisms, they are shielded from “accidental” complexity through the use of generative techniques and automation. This manifests itself in the *tool builder*’s approach of this thesis, as contributions in these thesis are supported by prototype implementations.

1.5 Outline of the Thesis

The thesis is organised as follows. Chapter 2 presents an approach to support the specification of temporal properties, as well as their verification to answer research question *RQ 1*. Chapter 3 discusses evolution of DSMLs, and presents a solution for research question *RQ 2*. Chapter 4 answers the research questions *RQ 3* and *RQ 4*, by presenting a solution for the composition of DSMLs. Chapter 5 concludes this thesis.

Appendix A presents an *Elevator* implementation that will serve as a basis for comparison in Chapter 2. Appendix B lists all evolution operations and Appendix C presents a prototype implementation in *metaDepth*, both in de context of Chapter 3.

CHAPTER 2

Verification for Domain-Specific Modelling Languages

Abstract. In Domain-Specific Modelling (DSM) the general goal is to provide Domain-Specific Modelling Languages (DSMLs) for domain users to model systems using concepts and notations they are familiar with, in their problem domain. Verifying whether a model satisfies a set of requirements is considered to be an important challenge in DSM, but is nevertheless mostly neglected. We present a solution in the form of *ProMoBox*, a framework that integrates the definition and verification of temporal properties in discrete-time behavioural DSMLs, whose semantics can be described as a schedule of graph rewrite rules. Thanks to the expressiveness of graph rewriting, this covers a very large class of problems. With *ProMoBox*, the domain user models not only the system with a DSML, but also its properties, input model, run-time state and output trace. A DSML is thus comprised of five sublanguages, which share domain-specific syntax, and are generated from a single metamodel. Generic transformations to and from a verification backbone ensure that the domain user is shielded from underlying notations and techniques. We explicitly model the *ProMoBox* framework's process in the chapter. Furthermore, we evaluate *ProMoBox* to assert that it supports the specification and verification of properties in a highly flexible and automated way.

2.1 Introduction

In Domain-Specific Modelling (DSM) [74] the general goal is to provide means for domain users to model systems in their problem domain. Techniques such as metamodelling and model transformation enable users to create Domain-Specific Modelling Languages (DSMLs) that can be used by domain users. Current DSM techniques allow users to model at the domain level and simulate the model, optimise the model, transform it to other formalisms, synthesise code, generate documentation, etc.

Verifying whether a model satisfies its requirements is an important challenge in DSM [65], but is nevertheless mostly neglected by current DSM approaches. Verification has been achieved by translating models to formal representations. Logic-based formulas in formalisms such as Linear Temporal Logic (LTL) [162] and Computation Tree Logic (CTL) [58] are used to represent the temporal properties that need to be verified [171]. These temporal properties can be verified using *e.g.,* model checking techniques [34]. Currently, domain users need to have a profound knowledge of some logic to express properties. This violates the principles of MPM. As with design models, the level of abstraction for specification and verification tasks needs to be raised to the domain level, as domain users should not be exposed to underlying technologies [210]. In this sense, DSM should not only address modelling the design of a system, but also its properties, its environment, its run-time state, and its execution traces, which should all be modelled at the domain level, in their own DSML.

Some dedicated property DSMLs and tools have been defined. For example, a visual formalism called TimeLine, developed at Bell Labs, allows users to specify temporal constraints, which are automatically translated to LTL [185]. Such approaches result in very suitable tools and languages, but a high development effort is required. Moreover, in the context of DSM, a DSML is highly prone to change during the development cycle, thus additional effort is required to keep verification tools synchronised with the DSML.

Some modelling languages are of such a nature that they can be used with minor modifications for defining system designs as well as properties. The Mathworks' Simulink® language for specifying the behaviour of dynamic systems as block diagrams [128] can be reused for specifying properties, by simply adding an "assertion block" [127]. Although strictly speaking not a DSML, Petri nets [168], used to specify the behaviour of concurrent systems, can also be used to express temporal properties to which execution traces conform [166], *i.e.,* the execution trace of a system that is modelled as a Petri net. This provides an elegant solution, as the language, and possibly its semantics, can be re-used. This is not possible for all design DSMLs.

Other approaches offer visual and generic modelling languages for specifying properties [100], but do not offer a domain-specific syntax, which is essential in DSM. In conclusion, no general DSM approaches exist today that provide dedicated support for verification as part of the DSML engineering process.

Problem statement. No general DSM solution exists for specifying properties at a domain-specific level, matching the domain-specific level at which design models are specified. Furthermore, such domain-specific property specification requires a means to automatically verify whether a system design satisfies these properties.

Main research question.

RQ 1. Can we find a general MPM solution to support the specification of (temporal) properties, as well as their verification? This solution should include support for verifying these properties. The user should only be exposed to domain-specific concepts. The solution should weigh as little as possible on the lightweight DSM process, and should be an extension of the existing DSM techniques for design languages.

The scope of this work is providing support for the definition and verification of temporal properties for any discrete-time behavioural DSML, for which the semantics can be described as a schedule of graph rewrite rules. Thanks to the expressiveness of graph rewriting, this covers a very large class of problems. In the following, the notions of *verification*, *properties* and *DSML* are within this scope. More detailed, the following research questions are posed:

— *RQ 1.1.* What languages are needed at the domain-specific level for specifying and verifying properties?

— *RQ 1.2.* How can these languages be generated from a specification of a design DSML?

— *RQ 1.3.* How can the verification activity, in this case by model checking, that uses these languages, be automated?

— *RQ 1.4.* What effort is required to enable verification support for a traditional DSM solution?

— *RQ 1.5.* Does the use of the presented approach result in a similar model checking time as its non-domain-specific counterpart?

— *RQ 1.6.* What is the impact of changes of/to different components of the approach?

— *RQ 1.7.* What are the assumptions and limitations of the approach?

In this chapter, *ProMoBox* is presented, a DSML engineering framework that aims to pull up property specification and verification tasks to the domain level. The contribution of the *ProMoBox* framework can be split in three parts. Firstly, it supports generic languages for modelling all artifacts that are needed for specifying and verifying properties that matches the level of specification used for design models. Secondly, it includes a fully automated method to specialise and integrate these generic languages into a given design DSML. Finally, it provides a fully automated bi-directional mapping to a suitable verification backbone for model checking (temporal) properties. Flexibility and automation are key in *ProMoBox*. *ProMoBox* supports definition and verification of temporal properties for any discrete-time behavioural DSML, for which the semantics can be described as a schedule of graph rewrite rules.

This chapter introduces the *ProMoBox* framework using the elevator controller running example presented in Section 1.3. In fact, the running example is inspired by an elevator controller model used to illustrate model checking [134]. This chapter shows a mapping to the *Spin* model checker [89] and LTL as verification backbone in detail. An informal evaluation of the *ProMoBox* framework is presented, to determine whether the *ProMoBox* approach fulfills its promises of flexibility and automation. We evaluate the model checking time of the running example, review the modelling process of *ProMoBox*, and discuss the impact of changes to any of the *ProMoBox* components. This includes replacing the mapping to *Spin* and LTL by a mapping to *Groove* [169] and CTL, to determine whether the *ProMoBox* approach fulfills its promises of flexibility and automation.

The chapter sets the stage in Section 2.2, where the necessary background is given on the subjects of DSM, running example, process modelling with FTG+PM (see Section 1.2.4), temporal

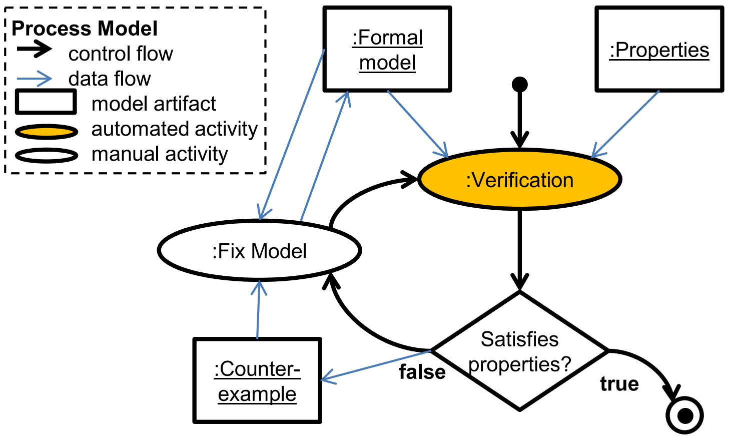


Figure 2.1: Verification of properties.

logic and model checking. In Section 2.3, the *ProMoBox* approach is presented. Section 2.4 discusses the mapping to a verification backbone. Section 2.5 shows how we applied *ProMoBox* to our Elevator DSML, and demonstrates how properties are checked and how a counterexample can be produced. In Section 2.6, the *ProMoBox* framework is evaluated. Section 2.7 discusses related work, and Section 2.8 concludes the chapter and gives some directions for future work.

2.2 Background

This section presents the prerequisites, additional to the background given in Section 1.2, to discuss the *ProMoBox* framework.

2.2.1 Temporal Properties

Properties allow us to encode questions we want to ask the modelled system, with an answer that is yes or no (*i.e.,* property satisfied or not). Examples are: (a) *Are there algebraic loops in my block diagram?*, or (b) *If I press a button in an elevator, will the elevator eventually reach the corresponding floor?* Useful properties are extracted from requirements, and it can be verified whether a model of a system *satisfies* these properties. The general verification process is shown in the Process Model (PM) of Figure 2.1, where a formal model and properties are fed into a verification engine. If no counterexample is found, the system satisfies the property. If a counterexample is found, it needs to be visualised in a domain-specific notation, so the user may correct the formal model, after which the verification process can be restarted.

We distinguish two kinds of properties: structural properties and temporal properties. *Structural properties* are evaluated statically on the structure of a model, *e.g.,* on the abstract syntax graph. Structural properties can be expressed in *e.g.,* a constraint language like the Object Constraint Language (OCL) [152]. Example (a) can be expressed in OCL. *Temporal properties* are defined in terms of time, *i.e.,* on paths of program execution, or execution *traces*.

One such way to express temporal properties is using Linear Temporal Logic (LTL) [162], as done in the *ProMoBox* approach. LTL is built up from propositions (*e.g.,* ψ and ϕ), on which the following temporal operators can be applied (yielding expressions that are also propositions): $\Box\psi$ (globally ψ), $\Diamond\psi$ (finally ψ), $\bigcirc\psi$ (next ψ), and $\psi\,\mathcal{U}\,\phi$ (ψ until ϕ). The logic operators $\vee$, $\wedge$, $\neg$ and $\rightarrow$ can be used in formulas, and operator precedence can be enforced by brackets. Throughout this chapter we will use $\psi\,\mathcal{W}\,\phi$ ("weak until") which can be defined as $(\psi\mathcal{U}\phi) \vee \Box\psi$. Knowing the syntax of LTL is not mandatory for understanding this chapter, but the interested reader can find a complete summary of the syntax and semantics of LTL in [57].

Another temporal logic is Computation Tree Logic (CTL) [58], that reasons about execution tree branches rather than single traces. Its syntax is similar to LTL, including the use of propositions, logic operators and brackets. It differs however in the temporal operators used: $AG\psi$, $EG\psi$, $AF\psi$, $EF\psi$, $AX\psi$, $EX\psi$, $A[\psi\,\mathcal{U}\,\phi]$, $E[\psi\,\mathcal{U}\,\phi]$ where the A in the operator means "for all paths", and E means "there exists at least one path". The G (globally), F (finally), X (next), and U (until) correspond to the LTL operators. Similarly to LTL, a "weak until" operator is defined as $A[\psi\,\mathcal{W}\,\phi] = \neg E[\neg\phi U(\neg\psi \wedge \neg\phi)]$ and $E[\psi\,\mathcal{U}\,\phi] = \neg A[\neg\phi W(\neg\psi \wedge \neg\phi)]$

The logic operators $\vee$, $\wedge$, $\neg$ and $\rightarrow$ can be used in LTL and CTL formulas.

Using LTL or CTL, temporal properties such as those in example (b) can be expressed. It is an example of a liveness property (something must eventually happen). The other class of temporal properties is the class of safety properties (something should not happen). There is a slight difference in verifying liveness and safety properties, as in order to find a counterexample in case of a liveness property, it must be shown that the proposition that must eventually happen is not present in traces with possibly infinite length.

2.2.2 Model Checking

Verifying whether a model satisfies a temporal property can be done in several ways. Some techniques for verification include manually inspecting the model, using testing techniques, symbolic execution, model checking, etc. In the *ProMoBox* approach we focus on model checking. Model checking is an automated approach, where it is determined whether a model satisfies a property by exhaustively searching for a counterexample, which takes the form of an execution trace. If no such counterexample is found, it is certain that the model satisfies the property.

Advantages of model checking include that it is a fully automated method, and that the outcome is reliable (unlike for *i.e.,* testing techniques). It has some major drawbacks however: the search space has to be finite, and large search spaces are infeasible to analyse. As the verification time as well as the required memory easily suffers from combinatorial explosion, the approach only allows a limited number of variables in the modelled system. This means that although model checking is in itself fully automated, often a manual abstraction step is required to make the technique applicable to a model [7]. Throughout this chapter, two tools for model checking will be used.

Spin [89] is a tool that allows users to write system descriptions in a textual programming language called *Promela*, encode properties in LTL, and verify whether the system description satisfies these properties. The default verification algorithm visits every possible state of the system description by building the state space explicitly. This is a directed graph with state vectors as nodes and statement executions as transitions. By traversing this complete state space a conclusive answer to the satisfaction of an LTL formula can be produced. *Spin*'s main application is the verification of concurrent systems. Although very powerful, writing system descriptions in *Promela* and LTL formulas requires a background in programming and logic, skills that domain users do not necessarily have. In our approach, we intend to leverage the power of *Spin*, without having to expose its technicalities (*i.e.,* its languages and interface) to domain users.

Groove[169] is a tool for specifying rule-based transformations on systems described as typed graphs. A type graph can be defined (much like a metamodel), and graphs can conform to the type graph. Similar to our view on model transformation, transformations consist of rules that, if matched, manipulate the graph. Rules are scheduled using an imperative language. *Groove* excels in its feature of verification of temporal logic properties written in LTL and CTL, which is typically not available in a domain-specific modelling tool. *Groove* is used in this chapter to evaluate the impact of replacing the verification backbone (*i.e., Spin*).

2.3 The Languages of *ProMoBox*

The state of the art in verification for DSM is illustrated in Figure 2.2. The diagram is divided into the domain layer where models are domain-specific, and the lower layer where artifacts are represented as logic formulas or code whose creation requires a significant theoretical background. A DSML has been developed, and systems can be modelled conform to the DSML. Transformations can be defined for simulating or translating this model, for generating documentation or platform-specific code. Note that these activities are not shown in Figure 2.2, as it exclusively focuses on the verification activity. Step 1 of Figure 2.2 consists of creating the formal model for verification. A transformation is shown that automatically compiles any model that conforms to the DSML to a formal representation (in this case *Promela*, a textual `pml` file), bridging the DSM layer and formal methods layer (step 1). While the DSML and model are usable by non-technical domain users, the necessary LTL-formulas (`ltl` files in the lower "formal methods" layer) for verification are not easily usable by domain users. In step 2 the formal model is executed in *Spin*, performing the actual model checking. In case of a counterexample, a `trail`-file representing the output trace is generated by *Spin*. This presents a second problem: understanding the trail also requires a similar theoretical background. In conclusion, the problem with the current state of the art in verification is that the DSM user still has to work in the formal methods layer. The goal of *ProMoBox* is to fully pull up the user experience, which was hitherto limited to the construction of system design models (and did not include requirements or property models), to the DSM layer, with a minimal increase in user effort.

This section presents the architecture of the *ProMoBox* framework. As *ProMoBox* builds on DSM, it consists of a language engineering phase and a system modelling phase. This section focuses on the language engineering phase. The system modelling phase is discussed in Section 2.5. The different models of the framework are discussed, and it is explained how the framework can be applied to a DSML, and how it relates to the existing model checking tool *Spin*. The *Elevator* case is used as a running example.

2.3.1 Outline of the *ProMoBox* Approach

The *ProMoBox* framework presented in this chapter builds on our earlier work [136, 138, 49]. *ProMoBox* stands for "Properties and (design) Models developed (Boxed) in concert". The language engineering support of *ProMoBox* consists of the following three parts.

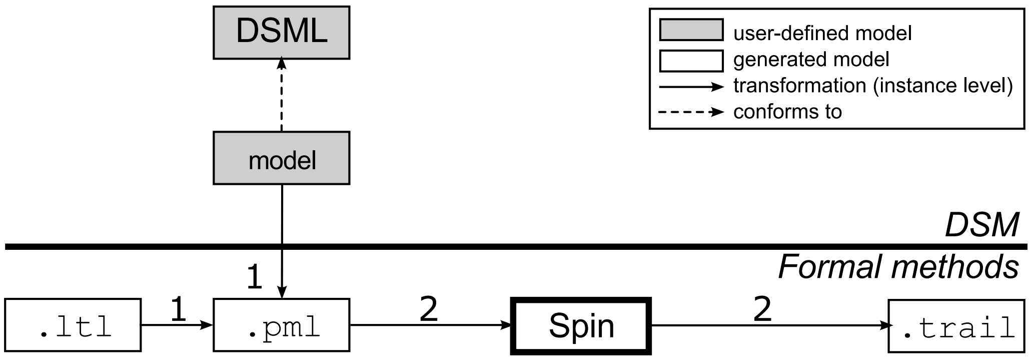


Figure 2.2: Verification in Domain-Specific Modelling.

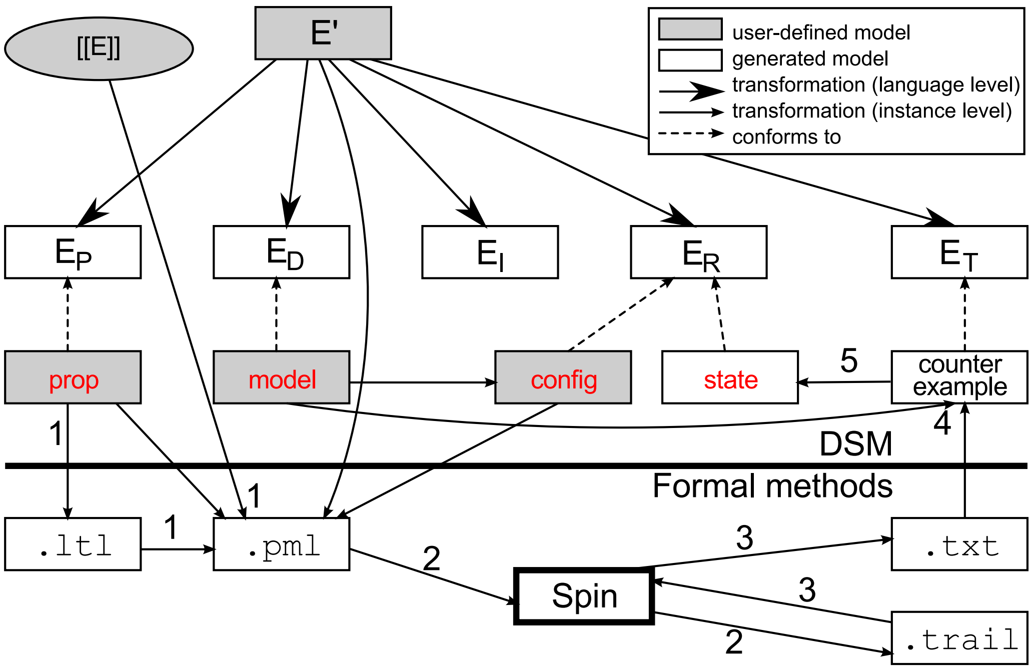


Figure 2.3: Property verification with *ProMoBox* and *Spin*.

The first part is the definition of five sublanguages. According to MPM principles everything should be modelled using the most appropriate formalisms. Therefore, *ProMoBox* replaces the traditional DSML with five sublanguages (each DSMLs) for modelling all artifacts that are needed to specify and verify properties [136]. The five sublanguages are the following:

— A *design language* for design modelling as supported by traditional DSMLs. With this language, the static structure (*i.e.,* language concepts that do not change at run-time) of the system is modelled. In case of Elevator, this includes all Elevator concepts, without the currentfloor association, nor the doors_open, going_up and pressed attributes;

— A *run-time language* for run-time state representation. The design language always includes all elements of the design language, plus dynamic state information that can change at run-time. Run-time instances are always associated with a design instance with the same static structure. One design instance possibly has multiple run-time instances corresponding with it, representing all possible states of the model. Note that in traditional DSM, the DSML often includes run-time concepts, meaning that no distinction is made between static structure and dynamic state. The running example was also presented including dynamic state information in Section 1.3. In fact, the metamodel shown in Figure 1.4 is the same as Elevator's run-time language;

— An *input language* to model event-based input (to model the environment in which the system operates). This language represents a stream of events. In *ProMoBox*, an input model is always an extensional stream of events. No explicit time delays are supported (as the LTL-based temporal properties also do not support this) but can be primitively emulated by adding empty events, each representing a time step. In case of Elevator, the input language only consists of a sequence of button presses;

— A *trace language* for state-based output representation (to model an execution trace of the system or verification counterexample). An execution trace is a sequence of run-time states connected with transitions that represent execution steps (*i.e.,* operational semantics' rule executions). The trace language can be used to represent execution traces of a simulation. A trace model is usually generated by a simulator or as a counterexample by a verification too. It can be generated manually as well for *e.g.,* modelling an oracle for a test case. In the Elevator case, this is a sequence (in terms of operational semantics steps) of run-time states with references to *e.g.,* move_up, open_doors, etc. (see Figure 1.7).

— A *property language* for property specification (to model temporal or structural properties). The properties language allows the user to define temporal properties, which are properties on the behaviour of systems. Properties are represented by temporal and structural operators over propositions. These propositions are patterns that can be matched or not matched (resulting in *true* or *false*) on a run-time state or static structure of a system. Since properties reason about state and model, all language constructs of the design language and run-time language are included in the property language. In the Elevator case, a property can express that whenever a button is pressed, the elevator should eventually reach the corresponding floor.

The second part is the generation of these five sublanguages. As the traditional DSML is replaced by five languages (*i.e.,* DSMLs), it would be time consuming to keep these intimately related sublanguages presented above consistent. Therefore, a fully automated method generates these sublanguages from a single DSML specification, keeping the five sublanguages consistent by construction. If necessary, simplifications are made in the DSML's metamodel, to address the scalability issues of model checking as described in Section 2.2.2. We extend metamodelling and model transformation languages with annotations, to add necessary information for every language construct and to introduce a conceptual simulation step. This additional information enables the fully automatic generation of the five sublanguages and necessary transformations between the sublanguages, thus minimising the effort of the language engineer. Each of the sublanguages is built from a DSML-independent template, and domain-specific language concepts. By using templates and a generative approach, the *ProMoBox* framework becomes configurable for various

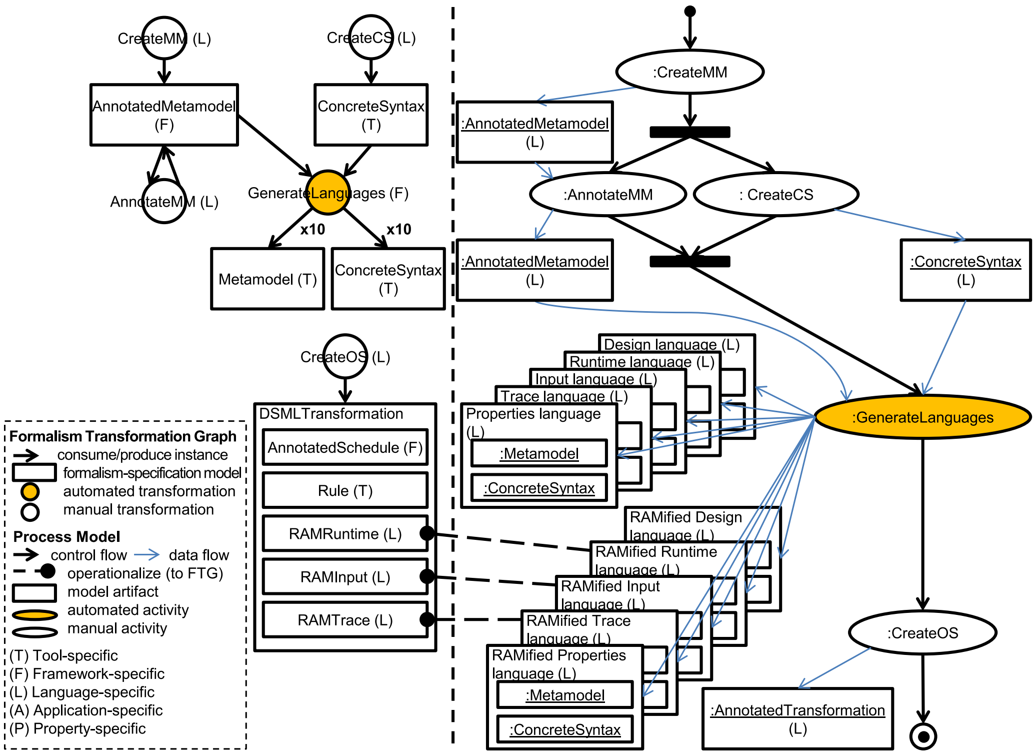


Figure 2.4: FTG+PM of language engineering with *ProMoBox*.

DSMLs.

The third part is the mapping to and from a verification backbone. A verification backbone based on the *Spin* model checker [89] is directly pluggable to DSM environments. Properties in *ProMoBox* are translated to LTL and a *Promela* model is generated that includes a translation of the domain-specific system specification, its initial state, the environment, and the rule-based operational semantics of the system. The verification results (in case of a counterexample) are translated back to the domain level. Note that *ProMoBox* assumes the correctness of the operational semantics, in order to adequately verify the system.

The Elevator *ProMoBox* is illustrated in Figure 2.3. When using the *ProMoBox* approach, only the grey models in Figure 2.3 need to be modelled by hand, the white models are generated. Note that only a minimal number of models needs to be created by hand thanks to the generative character of *ProMoBox*. All manually created models are at the DSM layer, meaning that modellers do not need to take a look “under the hood”. This is exactly the intent of DSM.

The remainder of this section explains language engineering with *ProMoBox* in detail. As the *ProMoBox* approach as presented in Figure 2.3 involves a significant number of modelling artifacts and transformation steps, the FTG+PM of Figure 2.4 serves as a process-oriented view on *ProMoBox* and will serve as a guide throughout this section. To put all artifacts in the FTG+PM (*i.e.*, languages, models and transformations) in perspective, they are marked with a specification level:

— (T) tool-specific: an artifact marked (T) (*e.g.,* a language for metamodelling) is defined by the DSM tool, and implemented by a DSM tool builder. As *ProMoBox* builds on DSM, it is a prerequisite for *ProMoBox*;

— (F) framework-specific: an artifact marked (F) (*e.g.,* a transformation generating the five sublanguages from an annotated DSML) is defined by the *ProMoBox* approach. One who aims to implement the *ProMoBox* framework will have to define this artifact, but once defined it can be used for any DSML *ProMoBox*;

— (L) language-specific: an artifact marked (L) (*e.g.,* annotating a metamodel) is defined by a language engineer when defining a DSML using *ProMoBox*;

— (A) application-specific: an artifact marked (A) (*e.g.,* the model shown in Figure 1.6) is defined by a domain user who instantiates the DSML to model a system. Since this section discusses the language engineering phase, application specific artifacts will only appear in Section 2.5 discussing the system modelling phase;

— (P) property-specific: an artifact marked (P) (*e.g.,* a model of a property or its counterexample) is defined by a domain user who models a property. Similar to application specific artifacts, property specific artifacts will only appear in Section 2.5.

The Process Model on the right side of the FTG+PM shown in Figure 2.4 starts with two manual traditional DSM tasks of creating a metamodel in the CreateMM activity as well as specifying a concrete syntax model (CreateCS). This results in an :AnnotatedMetamodel instance (rather than a Metamodel instance, yet still without annotations) and a :ConcreteSyntax instance. Then, the specific activities of the *ProMoBox*, further explained in the remainder of this section, are outlined. In Section 2.3.2, we define how metamodels can be annotated (the AnnotateMM activity, resulting in E' in Figure 2.3) to define the relationship with the five sublanguages. In Section 2.3.3, we discuss the sublanguages in detail and how they are generated from the annotated metamodel (the AnnotateMM activity). In Section 2.3.4, we present how to use these languages to create the annotated operational semantics which fine-tunes the behavioural semantics of the DSML *ProMoBox* (the CreateOS activity). In Section 2.3.5 we divert from the main FTG+PM track to give insight in how to migrate DSMLs, as created following the traditional process shown in Figure 1.2, to *ProMoBox*.

2.3.2 Defining a *ProMoBox*

In parallel with the :CreateCS activity, metamodel elements need to be annotated manually in the :AnnotateMM activity in Figure 2.4.

The metamodel elements (classes, associations and attributes) can be annotated with:

— *rt*: run-time, annotates a dynamic concept that serves as output (*e.g.,* a state variable);

— *ev*: an event, annotates a dynamic concept that serves as input and output (*e.g.,* a button press);

— *tr*: a trigger, annotates a static concept that also serves as input and output (*e.g.,* a key stroke – not available in the *Elevator* case).

These annotations are explicitly modelled as *annotation types* in the annotations model of Figure 2.7 which is shown in Figure 2.5. Technically, each annotation type represents *sublanguage*

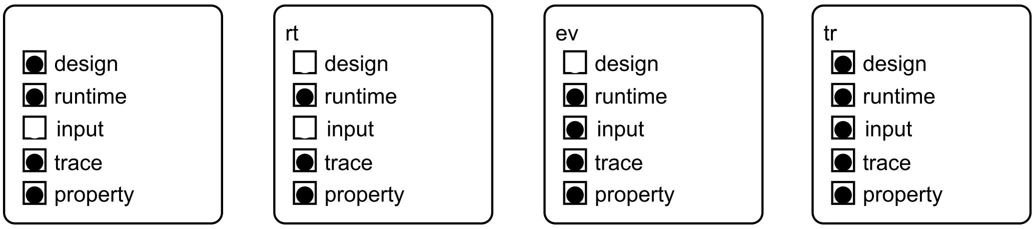


Figure 2.5: The annotations model of the *ProMoBox* framework.

membership, visualised by checked boxes in Figure 2.5. An annotation on a metamodel element denotes in which sublanguage the element becomes available (*i.e.,* included in the sublanguage's metamodel). For example, if an association is annotated with *rt*, it will be included in the run-time metamodel, the trace metamodel and the property metamodel, but not in the design metamodel and the input metamodel. We say for such an element, that it is (for example) *part of* the run-time metamodel. In case of annotations on an attribute, their containing class inherits the annotation's behaviour. Associations are treated similarly; their source and target classes inherit annotations if present. Subclasses inherit annotations from their superclasses.

The annotations model of Figure 2.5 can be extended with more annotation types, and will be automatically incorporated by the *ProMoBox* framework. An annotation type must adhere to the following rules:

— an annotation type should result in at least one inclusion in a sublanguage's metamodel;

— if an annotation type denotes inclusion in the design metamodel, then it must denote inclusion in the run-time metamodel as well;

— if an annotation type denotes inclusion in the input metamodel, then it must denote inclusion in the run-time metamodel as well;

— if an annotation type denotes inclusion in the run-time metamodel, then it must denote inclusion in the property metamodel as well.

The first annotation type of Figure 2.5 defines the case where no annotation is applied to a metamodel element. In that case, the concept is not part of the input language.

If multiple annotations are applied to an element, the union of all sublanguage memberships is taken. This union has to adhere to the above rules.

As shown in Figure 2.6, in the case of Elevator and its metamodel of Figure 1.4, all language concepts are static, except for the currentfloor association, and the doors_open, going_up and pressed attributes. The values of currentfloor, doors_open and going_up can change at run-time when the operational semantics transformation Figure 1.7 is applied. Therefore, the *rt* annotation is used, meaning that they will be part of the run-time, trace and property metamodel. The pressed attribute is annotated with *ev*, and can thus also serve as input. In fact, it will be the only metamodel element that appears in the input language.

As shown in Figure 2.4, from an annotated metamodel and a concrete syntax model, the fully automatic GenerateLanguages transformation generates a metamodel and a concrete syntax model for each of the five sublanguages, and for the RAMified sublanguages (so that the sublanguages can be used in transformations), resulting in a total of 10 languages.

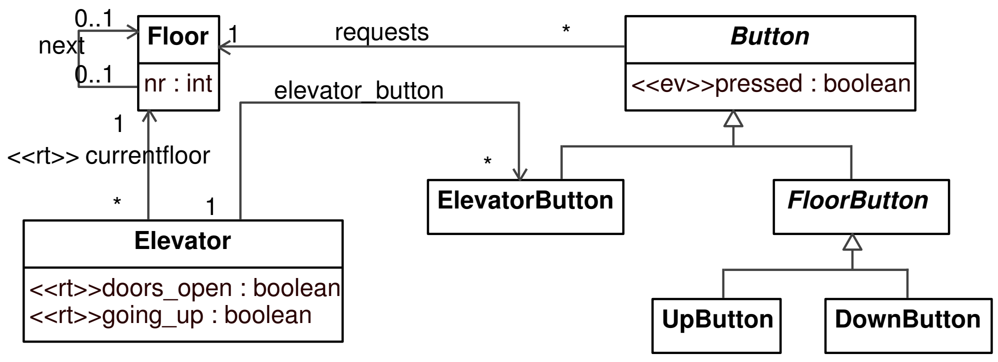


Figure 2.6: The annoted metamodel of the Elevator example.

The *ProMoBox* approach requires fully manual annotation of the metamodel. Afterwards the operational semantics are modelled. Nevertheless, the annotation phase can be partially automated by analysing the operational semantics model. This requires a different process than the one shown in Figure 2.6, where first the traditional modelling phase as shown in Figure 1.2 is completed, so that the operational semantics model is available before the annotation phase is started. The operational semantics' rules can subsequently be analysed to distinguish dynamic language elements from static language elements. Elements that change in the RHS of a rule are dynamic elements. However, it cannot be automatically determined whether a language element is part of the run-time language or part of the input language. Consequently, a manual annotation phase is still required. Following the process model shown in Figure 2.6, the annotations can be used to automatically define constraints on the transformation language of the operational semantics (or on any transformation language). Using these constraints, the user can be prohibited from changing static language elements in the transformation. The generation of these constraints can be incorporated in the RAMification process. This is however not included in the *ProMoBox* approach.

2.3.3 Generating the *ProMoBox* Languages

The five *ProMoBox* sublanguages are generated from an annotated metamodel and a concrete syntax model using a template-based approach. The fully automatic GenerateLanguages transformation of Figure 2.4 of one language and its RAMified counterpart is depicted in Figure 2.7.

Design, Run-time, Input and Output Languages

The design, run-time, input and trace languages are generated in a similar fashion. As for the abstract syntax, the metamodel is first filtered producing an ordinary metamodel in the FilterMM transformation. In this transformation, all language elements that are according to the annotations model not part of the to be generated language are removed, taking into account that annotations on attributes or associations are inherited by the related classes. From the elements that remain, remaining annotations are removed, so that the result is an ordinary metamodel.

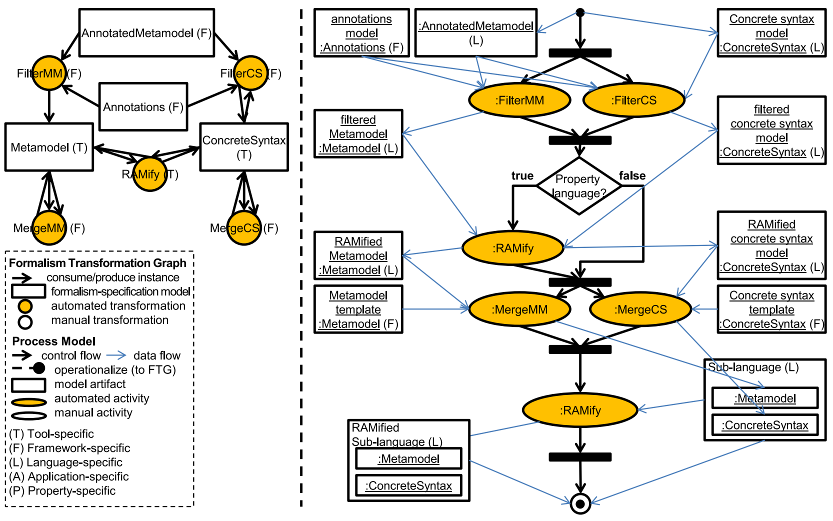


Figure 2.7: FTG (left) and PM (right) of generating one of the five sublanguages.

Then, depending on the to be generated language, a predetermined metamodel template is added to the metamodel in the MergeMM transformation. All templates have an Element class, with an attribute id, to which an inheritance relationship is created from all DSML-classes of the metamodel. This id will be used to link elements between different models. The result is the metamodel of the sublanguage.

Figures 2.8 through 2.12 show the generated metamodels of the Elevator DSML. The template elements are shaded. The template of the design language and the trace language consist of only one abstract *element*-class. The remainder of the metamodel are the DSML-specific elements, if they were annotated to be part of the language. This way, the dynamic elements currentfloor, doors_open, going_up and pressed do not appear in the design language E_d, but do appear in the run-time language E_r. The design instance of the elevator system with three floors is shown in Figure 2.9, from which dynamic concepts are excluded. In the input language, the template models an Environment as an Event list containing InputElements. In E_i, a series of inputs can consist of button presses. For now, we assume that at most one button can be pressed in the same event (the empty Event indicates that there is no input at that time). If the language engineer decides that more than one or exactly one button can be pressed at the same time, he can create a variant of this template (see also Section 2.6.3). The template of the trace language E_t consists of a *Trace* of *State*s and *Transition*s. This language is able to express a sequence of system states and the intermediate operations that caused the state change (rule_executions, rule applications in the operational semantics $E_{[[.]]}$, and/or an input events). The output of $E_{[[.]]}$, or the counterexample in verifications are instances of E_t. Due to the possibly large number of elements in such an execution trace, an instance of E_t is stored in textual form, and can be interpreted or “played out” by showing step-by-step an instance of the run-time language E_r.

The concrete syntax model of each of these languages is generated in a similar way. As depicted in Figure 2.7, the original concrete syntax model is filtered in the :FilterCS transformation. All icons and links of classes and associations that are not part of the to be generated metamodel are

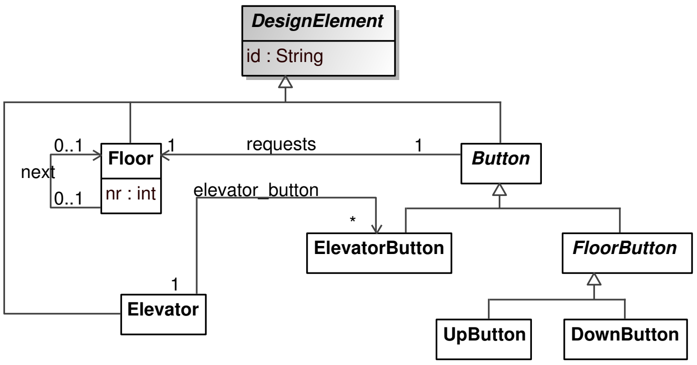


Figure 2.8: Metamodel of the Elevator design language E_d.

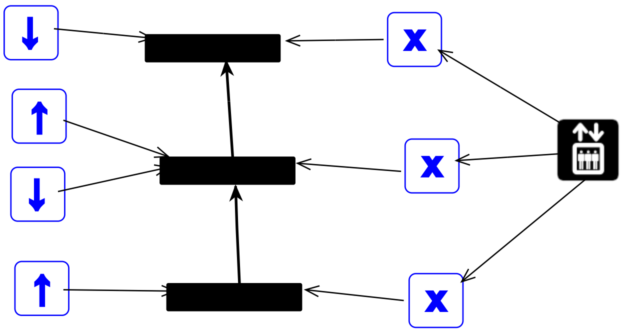


Figure 2.9: An instance of E_d modelling the design of Figure 1.6.

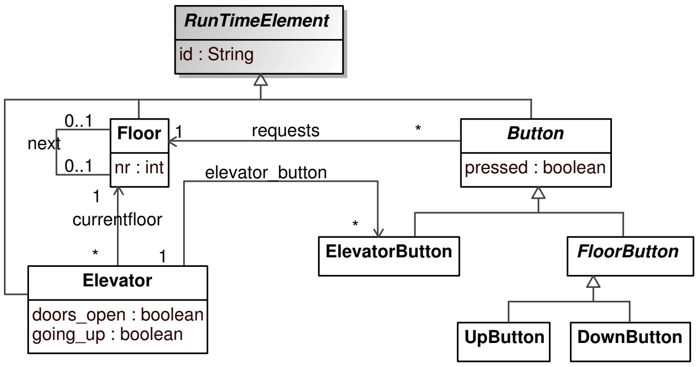


Figure 2.10: Metamodel of the Elevator run-time language E_r.

removed. Additionally, all concrete syntax elements such as text elements that contain references to attributes that are not part of the to be generated concrete syntax model are removed as well. Then, the template is added in the :MergeCS transformation, adding icons and links of respective template classes and associations. This results in a concrete syntax model, with a complete mapping to the

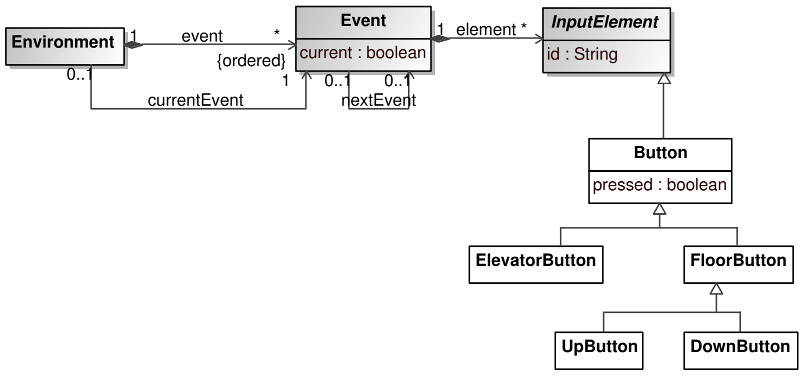


Figure 2.11: Metamodel of the Elevator input language E_i.

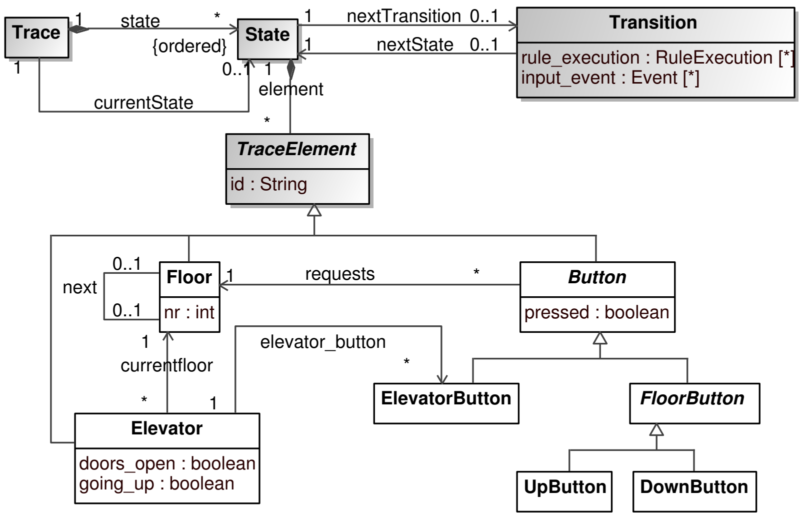


Figure 2.12: Metamodel of the Elevator trace language E_t.

abstract syntax model.

The result of the generation process for one sublanguage is an ordinary metamodel and concrete syntax model, thus fully compliant with the DSM tool.

Property Language

The property language E_p deserves special attention as it is the pivotal language of the *ProMoBox*, and its metamodel is generated in a slightly different way to the four other sublanguages, as shown in Figure 2.7 due to the decision node.

After filtering the metamodel according to annotations, an additional transformation (RAMification) is executed that produces a pattern language, suited for transformation languages [110, 193] as explained in Section 1.3. This language can however also be used to express structural patterns

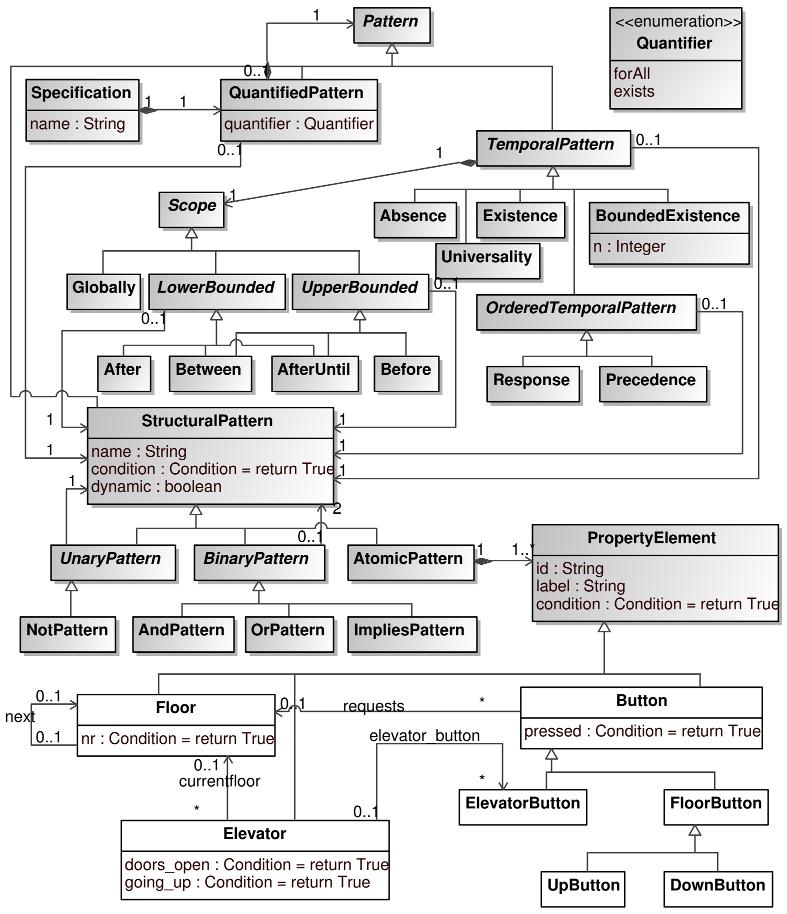


Figure 2.13: Metamodel of the Elevator Property Language E_p.

for properties as the same principle of pattern matching is re-used in this context. At the bottom of Figure 2.13 the RAMified DSML elements are shown. All attribute types are now conditions, abstract classes are now concrete classes, and lower bounds of multiplicities are relaxed to 0.

Next, the template for property languages is added to the metamodel, resulting in the metamodel of Figure 2.13. PropertyElement, the superclass of all DSML classes, includes a general condition, a label for inter-pattern matching similarly to [110, 193] and an id for inter-model traceability similar to the four other sublanguages. The template consists of a property Specification, which can be composed of the following three language constructs:

— *The quantification* of the formula by (i) *forAll* or *exists* clause(s), and (ii) corresponding structural pattern(s). The modeller can choose to model a property for all elements that match a given structural pattern. This structural pattern is evaluated on the design model, and can thus not refer to run-time concepts, because the match set must be static. Consequently, the property must be satisfied for all, or for one (depending on the quantifier) match(es) of the structural pattern. The resulting matches can be re-used as bound variables in the property, if they have

property	*scope*	*LTL formula*
Absence	Globally	$\Box(\neg\psi)$
Meaning:	Before ρ	$\Diamond\rho \to (\neg\psi\,\mathcal{U}\,\rho)$
ψ is false	After σ	$\Box(\sigma \to \Box(\neg\psi))$
Type: safety	Between σ and ρ	$\Box((\sigma \wedge \neg\rho \wedge \Diamond\rho) \to (\neg\psi\,\mathcal{U}\,\rho))$
	After σ until ρ	$\Box(\sigma \wedge \neg\rho \to (\neg\psi\,\mathcal{W}\,\rho))$
Existence	Globally	$\Diamond(\psi)$
Meaning:	Before ρ	$\neg\rho\,\mathcal{W}\,(\psi \wedge \neg\rho)$
ψ becomes true	After σ	$\Box(\neg\sigma \vee \Diamond(\sigma \wedge \Diamond\psi))$
Type: liveness	Between σ and ρ	$\Box(\sigma \wedge \neg\rho \to (\neg\rho\,\mathcal{W}\,(\psi \wedge \neg\rho)))$
	After σ until ρ	$\Box(\sigma \wedge \neg\rho \to (\neg\rho\,\mathcal{U}\,(\psi \wedge \neg\rho)))$
Bounded existence (2)	Globally	$(\neg\psi\,\mathcal{W}\,(\psi\,\mathcal{W}\,(\neg\psi\,\mathcal{W}\,(\psi\,\mathcal{W}\,\Box\neg\psi))))$
Meaning:	Before ρ	$\Diamond\rho \to ((\neg\psi\wedge\neg\rho)\,\mathcal{U}\,(\rho\vee((\psi\wedge\neg\rho)\,\mathcal{U}\,(\rho\vee((\neg\psi\wedge\neg\rho)\,\mathcal{U}\,(\rho\vee((\psi \wedge \neg\rho)\,\mathcal{U}\,(\rho \vee (\neg\psi\mathcal{U}\,\rho)))))))))$
ψ becomes true at most 2 times	After σ	$\Diamond\sigma \to (\neg\sigma\,\mathcal{U}\,(\sigma\wedge(\neg\psi\mathcal{W}\,(\psi\,\mathcal{W}\,(\neg\psi\,\mathcal{W}\,(\psi\,\mathcal{W}\,\Box\neg\psi))))))$
Type: safety	Between σ and ρ	$\Box((\sigma\wedge\Diamond\rho) \to ((\neg\psi\wedge\neg\rho)\,\mathcal{U}\,(\rho\vee((\psi\wedge\neg\rho)\mathcal{U}(\rho\vee((\neg\psi\wedge\neg\rho)\,\mathcal{U}\,(\rho \vee ((\psi \wedge \neg\rho)\mathcal{U}(\rho \vee (\neg\psi\,\mathcal{U}\,\rho))))))))))$
	After σ until ρ	$\Box(\sigma \to ((\neg\psi\wedge\neg\rho)\mathcal{U}\,(\rho\vee((\psi\wedge\neg\rho)\mathcal{U}(\rho\vee((\neg\psi\wedge\neg\rho)\mathcal{U}\,(\rho\vee((\psi \wedge \neg\rho)\mathcal{U}(\rho \vee (\neg\psi\,\mathcal{W}\,\rho) \vee \Box\psi))))))))))$
Universality	Globally	$\Box(\psi)$
Meaning:	Before ρ	$\Diamond\rho \to (\psi\,\mathcal{U}\,\rho)$
ψ is true	After σ	$\Box(\sigma \to \Box(\psi))$
Type: safety	Between σ and ρ	$\Box((\sigma \wedge \neg\rho \wedge \Diamond\rho) \to (\psi\,\mathcal{U}\,\rho))$
	After σ until ρ	$\Box(\sigma \wedge \neg\rho \to (\psi\,\mathcal{W}\,\rho))$
Precedence	Globally	$\neg\psi\,\mathcal{W}\,\phi$
Meaning:	Before ρ	$\Diamond\rho \to (\neg\psi\,\mathcal{U}\,(\phi \vee \rho))$
ϕ precedes ψ	After σ	$\Box\neg\sigma \vee \Diamond(\sigma \wedge (\neg\psi\,\mathcal{W}\,\phi))$
Type: safety	Between σ and ρ	$\Box((\sigma \wedge \neg\rho \wedge \Diamond\rho) \to (\neg\psi\,\mathcal{U}\,(\phi \vee \rho)))$
	After σ until ρ	$\Box(\sigma \wedge \neg\rho \to (\neg\psi\,\mathcal{W}\,(\phi \vee \rho)))$
Response	Globally	$\Box(\psi \to \Diamond\phi)$
Meaning:	Before ρ	$\Diamond\rho \to (\psi \to (\neg\rho\,\mathcal{U}\,(\phi \wedge \neg\rho)))\,\mathcal{U}\,\rho$
ϕ responds to ψ	After σ	$\Box(\sigma \to \Box(\psi \to \Diamond\phi))$
Type: liveness	Between σ and ρ	$\Box((\sigma \wedge \neg\rho \wedge \Diamond\rho) \to (\psi \to (\neg\rho\,\mathcal{U}\,(\phi \wedge \neg\rho)))\,\mathcal{U}\,\rho)$
	After σ until ρ	$\Box(\sigma \wedge \neg\rho \to ((\psi \to (\neg\rho\,\mathcal{U}\,(\phi \wedge \neg\rho)))\,\mathcal{W}\,\rho)$

Table 2.1: List of temporal properties (adapted from [55]).

the same label. Quantification patterns can be nested, or can contain a temporal or structural pattern.

— *The temporal pattern*, based on Dwyer's specification patterns [55]. The available temporal patterns are listed in Table 2.1. The temporal pattern allows the user to specify a pattern over a given scope, *e.g.,* "the absence of an occurrence of ψ, after the occurrence of σ", or "an occurrence of ψ is responded to by an occurrence of ϕ, between occurrences of σ and ρ" (with proposition variables ψ, ϕ, σ and ρ). Over 90% of the properties that were investigated by Dwyer et al. can be expressed in this simple framework [55]. Six patterns are supported, to express the absence, existence, bounded existence, universality response or precedence for given proposition(s). Additionally, a scope can be defined: is the pattern applicable globally, or after, before, in between or after until the occurrence of given proposition(s). The patterns and scopes are shown as template classes in Figure 2.13. In total up to four proposition variables can be used in a temporal pattern, and we implement them as structural patterns, that represent patterns on the state of the system at run-time. Table 2.1 shows each pattern and its type: *safety* or *liveness* property. A safety property is a property where something *should not* happen, and a liveness property is a property where something *should* happen eventually. The latter is verified in a different way, as such properties could be violated when the system can enter a loop that excludes the proposition that should happen. The LTL formula is shown for every temporal pattern. Some of them can be confusing to come up with manually, *e.g.,* existence of ψ before ρ, in which the case where ρ never occurs should be incorporated in the formula. Other LTL formulas are long and contain a high parenthesis depth, such as the bounded existence patterns. Note that these formulas do not yet include quantification and structural patterns, so they can easily end up being more complex.

— *The structural pattern*, based on PaMoMo [79, 80], for both design (when used in a quantification pattern) as well as run-time (when used in a temporal pattern) models. Only a small part of PaMoMo's expressiveness is included in the property language, but this suffices for defining most properties. The basic pattern is an AtomicPattern, which contains the DSML elements. Just like PropertiesElement, a StructuralPattern, can hold a condition, which returns *true* by default. Note that two types of structural patterns exist: static patterns that define patterns on a design model and dynamic patterns that define patterns on a run-time model. The former are used in quantifier patterns, the latter are used in temporal pattern propositions. The distinction is made using the dynamic attribute. This could have been modelled explicitly by adding StaticStructuralPattern and DynamicStructuralPattern as subclasses of StructuralPattern, each referring to their respective DSML elements. Since this would require duplicating the structural pattern elements and DSML elements, a constraint in metamodel of E_p enforces the use of correct patterns instead.

Apart from being RAMified, the concrete syntax model is generated in a similar way as the other sublanguages. The concrete syntax template for property languages consists of a combination of natural language and containers denoting visual, DSML patterns. The language engineer could choose to change the concrete syntax according to his or her preference to *e.g.,* icons denoting the temporal patterns, by changing the concrete syntax model.

Figure 2.14 shows a property for the Elevator DSML. The visual, domain-specific syntax is very similar to the concrete syntax of the rules in Figure 1.7. It is combined with a textual syntax

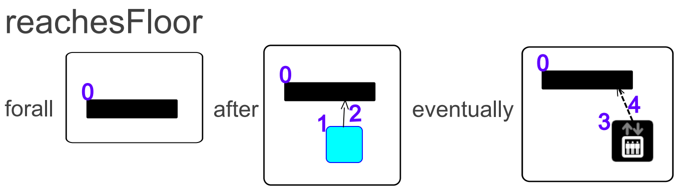


Figure 2.14: The reachesFloor instance of E_p: for any floor, after a button is pressed, the elevator will eventually open its doors on that floor.

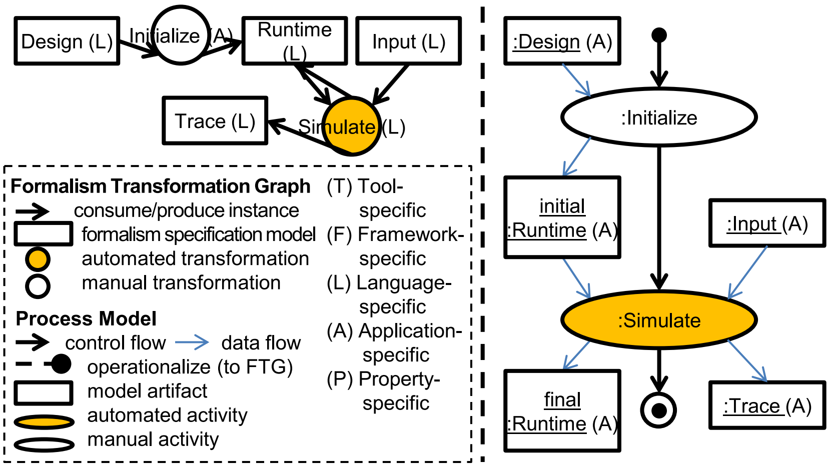


Figure 2.15: FTG+PM of operational semantics usage with *ProMoBox*.

to define temporal operators. A quantification pattern is used to denote that this property must be valid for all matches of the pattern, *i.e.,* for all floors. Its structural pattern (visualised by a container) binds a Floor of the design model of Figure 2.9 to the pattern element with label 0. On the right an existence pattern (the Elevator will reach the previously bound Floor) with lower temporal boundary (after a Button is pressed that requests the previously bound Floor) is shown.

2.3.4 Operational Semantics Annotation

ProMoBox makes input and output explicit in its sublanguages. Whereas the traditional Simulate transformation of Figure 1.2 has a DSML instance as input (*i.e.,* the model in its initial state), and produces a DSML instance (*i.e.,* the model in its final state), the Simulate transformation in *ProMoBox* takes a run-time model (initial state, generated or manually created from the design model) and an input model (input events) as input, and generates a run-time model (the final state) and a trace model (simulation trace) as shown in Figure 2.15. The transformation rules are restricted in that they can only change run-time elements of the model, and can only use design elements for matching. This restriction guides the language engineer in creating a correct model of the operational semantics. Moreover, since information about input and output needs to be incorporated in the semantics of the language, the operational semantics of Figure 1.7 need to be annotated. More specifically, the operational semantics have to define *when* (*i.e.,* at what point

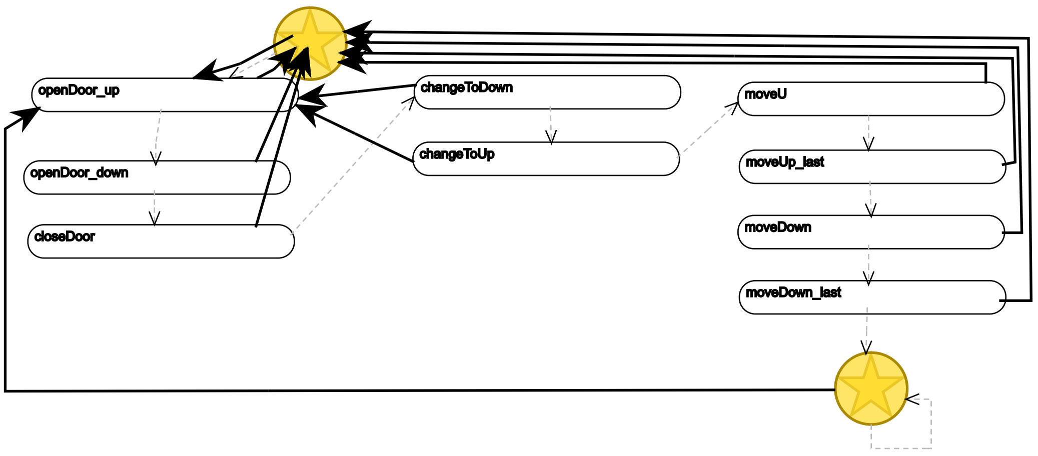


Figure 2.16: Annotated rule schedule of the operational semantics of the Elevator example.

in the rule schedule) a new input event is applied to the system, and a new state is added to the execution output trace. Different semantics are possible.

Perhaps the most straightforward semantics would be to read an event from the input environment and add the state to the output trace, following each successful rule execution of the operational semantics. This however implies a direct correlation between the rules of the operational semantics and a conceptual "time step" (*i.e.,* an abstraction of a significant period of time) of the semantics, while this is not necessarily so. In the Elevator example, it makes sense to consider moving up or opening the doors to be interpreted as a conceptual time step, but maybe not changing the direction, as this happens on a much smaller time time-scale than the elevator movement. Time-scale abstraction is hence appropriate [145]. Also, a more complex conceptual step might need more than one rule to express it, causing semantically inconsistent states in between. For example, the rule that opens the doors also immediately unlights the corresponding button. Suppose this would be modelled in two rules, then the language engineer might want to avoid adding the state to the trace that is in between those two actions.

To be able to distinguish between rules we introduce annotations for rule schedules, that allow determining the points when input is read and output is written. The CreateOS activity of Figure 2.4 includes the traditional modelling of the operational semantics and an annotation of the rule schedule. Two new language constructs are used in Figure 2.16: the input star and the Boolean attribute conceptualStep that allows full or dashed transitions.

When the schedule arrives in an input star, the next input event is read, and the currentEvent link (see Figure 2.11) is pointed to the next input event, if available. The input star thus models the interleaving between input language and run-time language. This cooperation between heterogeneous models is modelled using principles of semantic adaptation (which is extensively discussed in Section 4.3), where data (in this case button presses), time (in this case the conceptual time steps) and control (in this case determined by the rule schedule) of one formalism is adapted to the other [24, 135]. The input star can be translated to a generated rule implementing this. The rule links all elements in the input event by id to elements of the run-time model and propagates changes to the run-time model. The corresponding element in the run-time model can always be found because all elements of the input model are also present in the run-time model. Note that using metamodel annotations on classes would require the input language to be changed in order to be of full use. Because dynamically created class instances would not be addressable by id, the events should consist of patterns to indicate what part of the model receives input (*i.e.,* what button is lit). To that end, the DSML elements in the input language would have to be RAMified similarly to the property language, and structural pattern elements might have to be added. We choose not to do this to keep the input language simple, and because there are no additional limitations to our approach. After executing the input star, the successful link is modelled if the read event was not empty, or the notApplicable link if the event was empty.

Transitions in an annotated rule schedule can have as a side-effect that the current state is appended to the output trace (transition arrow with full line), or not (transition arrow with dashed line). By default, notApplicable transitions are dashed and successful transitions are full. Appending the current state to the output trace can be implemented as a rule Append that can be generated from the annotated metamodel. A transition with a full line can be translated to a traditional transition by replacing it with a transition of the same type to Append (a new reference to Append for each transformed transition), followed by a notApplicable and successful link to the target. The translation from an annotated transformation model to a traditional transformation model is defined as a higher order transformation, that serves as the translational semantics of the annotated transformation.

In the example of Figure 2.16, output is written after every successful rule application, except after reading input. Input is read after every successful rule application, except for the rules that change the direction. Note that the bottom input star keeps reading input events until something has changed (*i.e.,* a non-empty event was applied).

2.3.5 Migrating to *ProMoBox* Compliance

Throughout this section, we have assumed that the language engineer can start modelling a language from scratch. In this section, we will focus on how to simplify an available, traditionally modelled DSML and how to migrate this DSML to make it compatible with the *ProMoBox* approach.

Simplification

In some cases, the metamodel needs to be simplified manually to address scalability issues of model checking. We investigated this simplification step by applying the *ProMoBox* approach to a DSML for gestural interaction [49]. Together with annotating the metamodel and operational semantics model, this is the only manual task that needs to be performed in comparison with traditional DSML language design. As explained in Section 2.2.2, it is common practice to improve efficiency in model checking by reducing the search tree by abstracting the model [7]. To this end, we limit annotated metamodels with the following constraints:

— **attribute abstraction**: strings and floating point variables cannot be used as dynamic variables, although they can be used as static variables. Such variables need to be simplified to Booleans or integers, to decrease the domain size of the variable. Composite types, such as lists or maps, are not supported. The user may choose to model a number of attributes that represents an array, but this is not recommended as the number of variables is best kept as small as possible;

— **integer limitation**: integer values are restricted to a maximum of $[0..255]$ (byte in *Promela*) or $[-2^{15} - 1..2^{15} - 1]$ (short in *Promela*), depending on the user's choice when compiling the model;

— **association limitation**: associations in the metamodel can be instantiated a maximum number of times, depending on the user's choice when compiling the model;

— **dynamic class limitation**: similarly, classes in the metamodel (*i.e.,* classes must be part of the design metamodel) can be instantiated a maximum number of times.

Note that these constraints make the number of distinct states of the DSML (*i.e.,* instances of its run-time language) finite. This is essential in order to be able to analyse the entire state space, as done by explicit-state model checkers such as *Spin*.

Although the above constraints guarantee a finite state space, its size should be limited as much as possible to keep the memory usage low and execution time short when executing *Spin*'s model checking algorithm. In addition to adhering to these constraints, following some guidelines has a direct impact on the performance:

— **avoid dynamic classes**: it is not recommended in *ProMoBox* to include dynamic instantiation of classes in the DSML as it severely reduces efficiency by increasing the state space as *Promela* does not support dynamic instantiation;

— **limit variables**: it is good practice to limit the number of variables, especially integer attributes and dynamic associations (*i.e.,* that are not part of the design metamodel);

— **limit inheritance**: the number of inheritance links should be limited if possible, as inheritance needs to be emulated in *Promela*. Flattening the inheritance hierarchy reduces the execution time, because for pattern instances, less candidates have to be evaluated;

— **limit input language**: since model checking will take any possible scenario into account, the execution time is heavily dependent on what input it can get. Consequently, the number of distinct input scenarios should be limited by making abstractions that decrease the complexity of the input language (*i.e.,* limit the number of elements that are part of the input metamodel). When possible, it is recommended to translate integers in the input language to enumerations, which is an available data type in *AToMPM*.

The operational semantics transformation can be simplified in a similar way. There are no constraints in the design of the operational semantics, but some guidelines should be taken into account:

— **determinism**: it is recommended to avoid nondeterminism in the transformation's rule schedule, as nondeterminism can drastically increase the size of the state space;

— **pattern size**: the pattern size of each rule should be kept as minimal as possible as large patterns increase the state space depth;

— **number of pattern matches**: the number of matches that a pattern is expected to have should be kept to a minimum as a large number of matches increases the state space width. In particular, abstract classes in a pattern should be used with caution. One way of keeping the number of pattern matches low is using larger patterns (note that this is in conflict with the guideline above), and/or the use of negative application conditions (NACs) in patterns.

These simplifications only affect the DSML formalism. Note that the eventual model checking performance will also be heavily affected by the size of the system model and the complexity of the property.

Migration of Existing Models

A traditional metamodel can be automatically converted to an annotated metamodel. The metamodel language (*i.e.,* Class Diagrams) is a subset of *ProMoBox*'s metamodelling language Annotated Class Diagrams language, which only adds the possibility to annotate metamodel elements with different predefined annotations. We can analyse the migration impact of changes by interpreting switching from Class Diagrams to Annotated Class Diagrams as a language evolution (see Chapter 3). In this case, only additive, non-breaking changes are made. This means that the conformance of the Class Diagram instances (*i.e.,* the actual metamodels), is not *broken* after migrating them to Annotated Class Diagrams, only there are no annotations used. Depending on the tool that is used, it might however be necessary to change the namespace of the types used in the metamodel to explicitly denote that the metamodel is an annotated metamodel. This can be done using a simple transformation that traverses all metamodel elements and changes the type. Once the metamodel is migrated, the regular *ProMoBox* process can continue from AnnotateMM activity in Figure 2.4.

A traditional operational semantics model can be automatically converted to an annotated transformation model as well. Similarly, this is an additive, non-breaking change in the transformation language, where the rule schedule is migrated to an annotated rule schedule, and the patterns in rules are migrated to the run-time language. The migrated rule schedule will not yet include input stars, and will have the default output behaviour. The migrated rules do not change (except for namespace changes if necessary), and can be used in *ProMoBox* as is. Note that in case of an existing traditional operational semantics model, the metamodel annotation can be semi-automated by analysing the rules as explained in Section 2.3.2. This is however out of scope of this thesis.

An existing concrete syntax model does not need to be migrated as traditional concrete syntax modelling is still used in the *ProMoBox* approach.

As a consequence of the migration of a traditional metamodel, existing instances should be converted as well. The traditional metamodel might include static, dynamic and input concepts, which are all included in the run-time language according to the annotation type rules. This means that to migrate instances to the run-time language, yet again, no explicit migration is necessary. If however the instance has to be migrated to the design language or input language, some language constructs might become unavailable. These metamodel changes can be calculated in retrospect by analysing the difference between the traditional metamodel and the design or input metamodel. The resulting metamodel changes are eliminate metaclass, and eliminate metaproperty (which covers both associations and attributes) (see Chapter 3). These are subtractive, breaking and resolvable changes, so a corresponding migration transformation can be generated. The transformation will remove all instances of removed meta-elements, thus again establishing conformance with the target language (*i.e.,* design or input language).

As a conclusion, all existing DSM artifacts can be automatically migrated to *ProMoBox*, so that the additional required effort for the language engineer is limited to annotating the metamodel and transformation schedule.

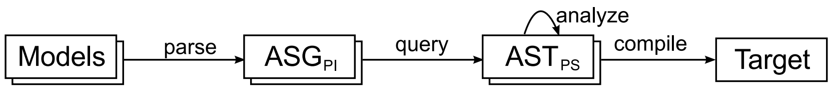


Figure 2.17: The *ProMoBoxPromela* compilation process.

2.4 Mapping to a Verification Backbone

As shown in Figure 2.3, a generic transformation generates a *Promela* model (a `pml` file) containing an LTL formula by means of a model-to-text transformation. The *Promela* model is compiled from the following models:

— the annotated metamodel, that is used to generate the *Promela* run-time metamodel and environment;

— an initial state, a run-time model;

— a property model;

— an operational semantics model;

— the annotations model (not shown).

A dedicated *Promela* model is generated that is optimised for a specific modelled system, initial state and property. The compilation process follows the steps shown in Figure 2.17. First, the models are parsed following the traditional concrete syntax parsing, resulting in a platform independent abstract syntax graph ASG_{PI}. Then, the abstract syntax is queried taking the *Promela* target language into account to obtain a platform specific abstract syntax tree AST_{PS}. The AST_{PS} is used to perform semantic analysis that organises the tree to generate optimised *Promela* code. Finally, the AST_{PS} is compiled to *Promela* including an LTL formula. The compilation results in a `pml` file that serves as input for the *Spin* verification tool.

Promela is a language specifically designed for explicit state model checking, meaning that during model checking the state space, *i.e.,* a graph of states of the system, is formed. This means that in designing a suitable *Promela* model, certain restrictions have to be taken into account:

— the *state space* (*i.e.,* number of different states and transitions) needs to be as small as possible. This means that nondeterminism has to be limited to constrain the breadth of the state space and that the number of atomic statements has to be limited to constrain the depth of the state space. This influences the verification time and memory consumption;

— the *state vector* has a static size and needs to be a small as possible, so that the individual nodes in the graph are as small as possible. This mainly influences the memory consumption of the verification.

The overall structure of the `pml` file is shown in Listing 2.1, where code snippets are referenced between angle brackets, which are explained in detail below. The *Promela* model is structured as follows, and the role of each model in the compilation process is discussed in the remainder of this subsection:

— line 1: code for the metamodel (see Section 2.4.1) which implements the definition of types;

— line 2-3: declaration of the system variables for the static `s` and dynamic part `r` of the system,

```
 1 <METAMODEL>
 2 hidden __System s;
 3 __RuntimeSystem r;
 4 <ENVIRONMENT>
 5 <LTL FORMULA>
 6 active proctype instance() {
 7   <INSTANCE>
 8   <SET INITIAL RULE>
 9   do ::
10     atomic {
11       <RULE SCHEDULE>
12       <RULE 1>
13       <RULE 2>
14       ...
15       <RULE N>
16       <PRINT STATE>
17       <UPDATE STATE>
18     }
19   od;
20 }
```

Listing 2.1: The overall structure of the generated *Promela* model.

which is typed by the metamodel. The `hidden` keyword denotes that `s`, the static part of the system, is not part of the state vector;

— line 4: code for the input language (see Section 2.4.3) which is implemented as a function that nondeterministically executes an input event;

— line 5: an LTL formula implementing the temporal pattern (see Section 2.4.5);

— line 6: start of the main process (stops at line 20);

— line 7: code for the initial state (see Section 2.4.2) initialising the system variable by assigning values to fields of `r` and `s`;

— line 8-15: code for operational semantics (see Section 2.4.4), including a `do`-loop, *i.e.,* the simulation loop, that applies a rule each indivisible atomic iteration (ends at line 19). This includes a rule schedule, which branches (by using `goto`s) to code for rules;

— line 16: code for the trace language (see Section 2.4.6), which prints out the current state of the system in a predefined format, which can be written to a text file;

— line 17: code for evaluating the property propositions (*i.e.,* structural patterns) which will be executed after every state change (*i.e.,* rule application) (see Section 2.4.5).

2.4.1 Compilation of the Metamodel

A metamodel is created in the form of `typedef` statements as shown in Listing 2.2, that allow the declaration of statically structured types. First, an `mtype` declaration is given, which introduces symbolic names for concrete metamodel types and rules (the latter will be used in Section 2.4.4). Only the three classes on top of the inheritance hierarchy become *Promela* types (line 2-26). The instances of these types are stored as static arrays, and instances are accessed by indexing that array. Attributes are converted to `typedef` fields, with corresponding types. Since *Promela* is not an object-oriented language, inheritance and associations have to be encoded, as shown in

```
1  mtype = {movedown_last_, opendoor_up_, changetoup_, changetodown_, opendoor_down_, movedown_, moveup_last_, closedoor_,
       environmentstep2_, environmentstep1_, moveup_, ElevatorButtonType, ElevatorType, FloorType, UpButtonType,
       DownButtonType}
2  typedef Button {
3    short __subtype;
4    short requests_out; // Floor out
5    short elevator_button_in; // Elevator in
6  }
7  typedef RuntimeButton {
8    bit pressed;
9  }
10 typedef Elevator {
11   short __subtype;
12   short elevator_button_out[3]; // ElevatorButton out
13 }
14 typedef RuntimeElevator {
15   bit doors_open;
16   bit going_up;
17   short currentfloor_out; // Floor out
18 }
19 typedef Floor {
20   short __subtype;
21   short nr;
22   short next_out; // Floor out
23   short next_in; // Floor in
24   short requests_in[3]; // Button in
25   short currentfloor_in; // Elevator in
26 }
27 typedef __System {
28   Button button_[7];
29   Elevator elevator_[1];
30   Floor floor_[3];
31 }
32 typedef __RuntimeSystem {
33   RuntimeButton button_[7];
34   RuntimeElevator elevator_[1];
35 }
```

Listing 2.2: The compiled bounded meta-model.

Listing 2.2. Inheritance is implemented by the `__subtype` attribute, that refers to any class in the run-time or design metamodel. Associations are implemented with bidirectional navigability by fields of type `short`, that refer to the index of the target, rather than to an object. For instance, if the `currentfloor_out` of an `Elevator` is 1, its target is the `Floor` with index 1. If no link exists, its index is set to -1. Two kinds of types are created: static types (`Button`, `Elevator`, `Floor` and `__System`) that model the design language, and dynamic types (`RuntimeButton`, `RuntimeElevator` and `__RuntimeSystem`) that model the additional elements of the run-time language. This distinction is made so that the state vector only contains a minimum of state information.

The model of the initial state of the modelled system (modelled as a run-time instance) is taken into account in three ways to make sure the state space of the *Promela* model is bounded. First, on line 27-35, a `__System` type is declared with static arrays of 7 Buttons, 1 Elevator and 3 Floors, and a `__RuntimeSystem` type with 7 Buttons and 1 Elevator, referring to the multiplicities of the design model. These two types represent the static and dynamic part of the modelled system, and both are instantiated once as shown in Listing 2.1 on line 2-3. Second, maximum array sizes for fields implementing associations are chosen according to the multiplicities of the design model. If the number of possible instances of a dynamic association is unbounded, a maximum number is set to ensure boundedness of the *Promela* model. Third, only one end of each dynamic association

```
d_step {
  // ElevatorButton $atompmId:1
  s.button_[0].__subtype = ElevatorButtonType;
  r.button_[0].pressed = true;
  s.button_[0].requests_out = 0;
  s.button_[0].elevator_button_in = 0;
  // ElevatorButton $atompmId:2
  s.button_[1].__subtype = ElevatorButtonType;
  r.button_[1].pressed = false;
  s.button_[1].requests_out = 1;
  s.button_[1].elevator_button_in = 0;
  (...)

  // Elevator $atompmId:0
  s.elevator_[0].__subtype = ElevatorType;
  r.elevator_[0].doors_open = true;
  r.elevator_[0].going_up = false;
  s.elevator_[0].elevator_button_out[0] = 2;
  s.elevator_[0].elevator_button_out[1] = 0;
  s.elevator_[0].elevator_button_out[2] = 1;
  r.elevator_[0].currentfloor_out = 2;

  // Floor $atompmId:4
  s.floor_[0].__subtype = FloorType;
  s.floor_[0].nr = 2;
  s.floor_[0].requests_in[0] = 3;
  s.floor_[0].requests_in[1] = 0;
  s.floor_[0].requests_in[2] = -1;
  s.floor_[0].next_in = 1;
  s.floor_[0].next_out = -1;
  s.floor_[0].currentfloor_in = -1;
  (...)
}
```

Listing 2.3: The compiled design model and initial state (abbreviated).

needs to be stored in the dynamic state vector, and the one with the lowest multiplicity of the design model is chosen in order to limit the state vector size. Both ends are stored to improve navigability but will constantly be kept synchronised, so only one end has to be part of the state vector. In this example, the `currentfloor_out` field has a multiplicity of maximum 1 as there is only one Elevator in the design model, while the `currentfloor_in` field has a multiplicity of maximum 3 as there are three Floors, so the former is stored in the state vector.

2.4.2 Compilation of the Model and Initial State

The first statements of the one and only active process started on line 6 in Listing 2.1 set the values of the initial state of the modelled system as shown partially in Listing 2.3. The values for each model element are set, creating attribute values, type values and association links. In comments the corresponding type and element id from the modelling tool are given as documentation. This results in the static `__System` instance `s` and the initial state of the dynamic `__RuntimeSystem` instance `r`. The element order of array fields in `__System` and `__RuntimeSystem` is by definition the same, meaning that *e.g.,* `s.button_[2]` refers to the same Button instance as `r.button_[2]`. The initialisation code is in a `d_step` block, making it an indivisible, deterministic block of code.

```
1  inline environment() {
2    if
3      :: true -> skip
4      :: r.button_[0].pressed == 0 ->
5         d_step { success = true; r.button_[0].pressed = true; }
6      :: r.button_[1].pressed == 0 ->
7         d_step { success = true; r.button_[1].pressed = true; }
8      :: r.button_[2].pressed == 0 ->
9         d_step { success = true; r.button_[2].pressed = true; }
10     :: r.button_[3].pressed == 0 ->
11        d_step { success = true; r.button_[3].pressed = true; }
12     :: r.button_[4].pressed == 0 ->
13        d_step { success = true; r.button_[4].pressed = true; }
14     :: r.button_[5].pressed == 0 ->
15        d_step { success = true; r.button_[5].pressed = true; }
16     :: r.button_[6].pressed == 0 ->
17        d_step { success = true; r.button_[6].pressed = true; }
18   fi
19 }
```

Listing 2.4: The compiled environment model.

2.4.3 Compilation of the Input Language

Listing 2.4 shows a macro containing a model of one execution step of the environment that implements the input language. It represents passing through an input star during simulation. It consists of an if-statement that can set the `pressed`-value of any Button to *true* (or 1) if it was not yet *true*. *Promela* evaluates the if-statement by evaluating all of its conditions, then chooses randomly one option that evaluates to *true*, and executes the corresponding body. The code of an environment step shown above ensures that at most one, but also no (see line 3) button can be pressed. Other semantics can be chosen which would result in a variation of this macro, such as: exactly one option has to be chosen, more than one but unique options can be chosen, more than one and the same options can be chosen, etc. Also, in our current implementation, a Boolean input is implemented that can only be turned on, not off, by the environment. The environment strongly influences the size of the state space, as this macro is typically executed a multitude of times and might result in a significant number of state space branches. Therefore, the possible options (in this case maximally 8) should be kept to a minimum by wisely choosing the above described environment variant. This often requires an abstraction. The success variable is necessary to denote whether a rule was applied and reflects whether the success or notApplicable link exiting an input star should be followed in the rule schedule. The print statement will be used to create a textual report in the case of a counterexample (see Section 2.4.6).

2.4.4 Compilation of the Operational Semantics

Listing 2.5 shows the partial rule schedule of Figure 2.16 in *Promela*. This rule schedule is part of the simulation loop shown in Listing 2.1. In line 1-14, control flow is directed to the correct rule. The initial rule is set earlier in the code, right before the start of the simulation loop, by assigning the `nextrule` variable. From line 16 onward, the schedule is modelled that is activated after evaluating a rule. After evaluation, the rule was either successful or not applicable, which is modelled by the `success` variable that is set in the rule code. According to this, the new value of `nextrule` is determined, and depending on whether output must be added to the output

```
 1 if
 2   :: (nextrule == movedown_last_) -> goto MOVEDOWN_LAST;
 3   :: (nextrule == movedown_) -> goto MOVEDOWN;
 4   :: (nextrule == moveup_last_) -> goto MOVEUP_LAST;
 5   :: (nextrule == opendoor_up_) -> goto OPENDOOR_UP;
 6   :: (nextrule == closedoor_) -> goto CLOSEDOOR;
 7   :: (nextrule == changetoup_) -> goto CHANGETOUP;
 8   :: (nextrule == changetodown_) -> goto CHANGETODOWN;
 9   :: (nextrule == initialize_) -> goto INITIALIZE;
10   :: (nextrule == environmentstep2_) -> goto ENVIRONMENTSTEP2;
11   :: (nextrule == environmentstep1_) -> goto ENVIRONMENTSTEP1;
12   :: (nextrule == opendoor_down_) -> goto OPENDOOR_DOWN;
13   :: (nextrule == moveup_) -> goto MOVEUP;
14 fi;
15 
16 MOVEDOWN_LAST_schedule:
17 if
18   :: (success == true) -> // when successful
19   rule = movedown_last_;
20   nextrule = environmentstep1_;
21   goto OUTPUT;
22   :: else -> // when not applicable
23   nextrule = changetodown_;
24   goto UPDATESTATE;
25 fi;
26 OPENDOOR_UP_schedule:
27 if
28   :: (success == true) -> // when successful
29   rule = opendoor_up_;
30   nextrule = environmentstep1_;
31   goto OUTPUT;
32   :: else -> // when not applicable
33   nextrule = opendoor_down_;
34   goto UPDATESTATE;
35 fi;
36 (...)
37 ENVIRONMENTSTEP1_schedule:
38 if
39   :: (success == true) -> // when successful
40   nextrule = opendoor_up_;
41   goto UPDATESTATE;
42   :: else -> // when not applicable
43   nextrule = opendoor_up_;
44   goto UPDATESTATE;
45 fi;
46 
47 ENVIRONMENTSTEP2:
48 success = false;
49 environment();
50 goto ENVIRONMENTSTEP2_schedule;
51 
52 ENVIRONMENTSTEP1:
53 success = false;
54 environment();
55 goto ENVIRONMENTSTEP1_schedule;
```

Listing 2.5: The compiled rule schedule (abbreviated) and environment steps.

trace control flow is directed to `OUTPUT` (see Listing 2.1 line 16) or directly to `UPDATESTATE` (see Listing 2.1 line 18), after which a new iteration of the simulation loop starts. Note that a new iteration starts after every evaluation of a rule, successful or not.

The input stars are modelled as rules on line 47-55 and as such, can be triggered by line 1-14. The `success` variable is set to false, and can be set to *true* in the function call to the environment macro of Listing 2.4. Like any rule, control flow is directed to the right `schedule` block at the end

```
CHANGETOUP:
success = false;
e0 = 0; // this is the only Elevator so index must be 0
f1 = r.elevator_[e0].currentfloor_out; // direct access
b2_indexes[0]=0; b2_indexes[1]=1; b2_indexes[2]=2; b2_indexes[3]=3; b2_indexes[4]=4; b2_indexes[5]=5; b2_indexes[6]=6;
b2_max = 7;
do
:: (success == false && b2_max > 0) ->
  if
  :: ((e0 >= 0) && (s.elevator_[e0].__subtype == ElevatorType)
  && (r.elevator_[e0].going_up == false)
  && (f1 >= 0)
  && (s.floor_[f1].__subtype == FloorType)
  && (b2_max > 0) && (b2_indexes[0] >= 0)
  && (s.button_[b2_indexes[0]].__subtype == ElevatorButtonType
      || s.button_[b2_indexes[0]].__subtype == DownButtonType
      || s.button_[b2_indexes[0]].__subtype == UpButtonType)
  && (r.button_[b2_indexes[0]].pressed == true)
  && (s.button_[b2_indexes[0]].requests_out >= 0)
  && (s.button_[b2_indexes[0]].requests_out != f1)
  && (s.floor_[s.button_[b2_indexes[0]].requests_out].__subtype == FloorType)
  && (s.floor_[s.button_[b2_indexes[0]].requests_out].nr>s.floor_[f1].nr))
  -> b2 = b2_indexes[0]
   (...)
  :: else -> break
  fi;
  f4 = s.button_[b2].requests_out; // direct access
  // NAC_
  NAC__success = false;
  b7_indexes[0]=0; b7_indexes[1]=1; b7_indexes[2]=2; b7_indexes[3]=3; b7_indexes[4]=4; b7_indexes[5]=5; b7_indexes[6]=6;
  b7_max = 7;
  do
  :: (NAC__success == false && b7_max > 0) ->
    if
    :: ((b7_max > 0) && (b7_indexes[0] >= 0)
    && (s.button_[b7_indexes[0]].__subtype == ElevatorButtonType
        || s.button_[b7_indexes[0]].__subtype == DownButtonType
        || s.button_[b7_indexes[0]].__subtype == UpButtonType)
    && (r.button_[b7_indexes[0]].pressed == true)
    && (s.button_[b7_indexes[0]].requests_out >= 0)
    && (s.button_[b7_indexes[0]].requests_out != f1)
    && (s.floor_[s.button_[b7_indexes[0]].requests_out].__subtype == FloorType)
    && (s.floor_[s.button_[b7_indexes[0]].requests_out].nr < s.floor_[f1].nr))
    -> b7 = b7_indexes[0]
     (...)
    :: else -> break
    fi;
    f6 = s.button_[b7].requests_out; // direct access
    // NAC_ matched
    NAC__success = true;
    break;
  :: else -> break
  od;
  if
  :: (NAC__success == false) ->
    r.elevator_[e0].going_up = true;
    success = true;
    break;
  :: else ->
    b2_max—; temp = b2_indexes[b2_max]; b2_indexes[b2_max] = b2_indexes[b2_index]; b2_indexes[b2_index] = temp;
  fi;
:: else -> break
od;
goto CHANGETOUP_schedule;
```

Listing 2.6: The compiled changeToUp rule (abbreviated).

of the rule.

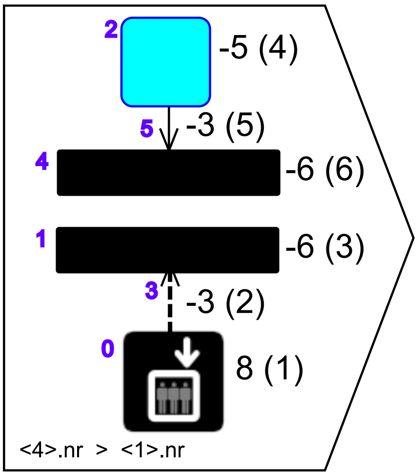


Figure 2.18: The order of compilation of the LHS of the changeToUp rule.

The compiled changeToUp rule is shown in Listing 2.5. Just like the input star, it starts by setting the `success` variable to false and ends with a `goto` statement. A rule consists of an LHS (the pattern that must be matched), optional NACs (given an LHS match, patterns that should not match) and an RHS (the effect of the pattern).

For each of these three parts, its pattern elements are compiled one by one. The order in which elements are compiled is determined by a sorting algorithm, that sorts according to a score for each element, representing the expected probability to find a match. This compilation order is illustrated in Figure 2.18, where the score of each element, followed by the order of compilation between brackets, is shown at the right side of each element on Figure 2.18. The score given to each pattern element is determined by the default formula: $10 \cdot nrOfAttributes - 2 \cdot multiplicity - subclassMatching$, with $nrOfAttributes$, the number of attributes with a constraint, $multiplicity$, the number of elements of that type, and $subclassMatching$ whether or not (value 1 or 0) this element can be matched with candidates of a subtype (*e.g.,* an UpButton can match a Button pattern element). Pattern elements with high scores are more constrained and thus less likely to match. For efficient pattern matching, it is desirable to know as soon as possible if the pattern will not match, and it is therefore preferable to try to match "hard" elements first. A different sorting algorithm can be plugged in easily. From a compiled element, the code generator traverses the pattern by following links from the element. A similar pluggable sorting algorithm is implemented for links to determine which of the pattern element links should be matched first.

In *Promela*, a match candidate is represented by an array index stored in a variable and the variable name is composed of the first letter of its type, followed by the label of the pattern element (*e.g.,* `e0` on line 3 represents a match candidate for the Elevator with label 0 in the changeToUp rule in Figure 1.7). In principle, the code for matching the elements and thus finding the right candidate consists of nested blocks in order of element traversal. Depending on the element one out of two kinds of code blocks is generated:

— a *do block*, if there are multiple candidates, *e.g.,* for matchingButtons. The do block iterates over all candidates until one is found that satisfies the element constraints. According to the pattern matching semantics, a random candidate has to be found. This is achieved by storing all valid candidates in a temporary `_indexes` array (line 5, 30). The first `_max` (line 6, 31) elements in the array are the candidates that are not evaluated yet. Since at the start no elements are evaluated yet, `_max` is equal to the length of `_indexes`. If a candidate does not satisfy the pattern element constraints, its `_indexes` entry is swapped and `_max` is decreased so that the candidate is not in the first `_max` elements of `_indexes` anymore (line 62). In each iteration, a candidate is chosen by an if statement that chooses a random candidate for which the additional conditions are met (line 9-26, 34-49). Each option has the same condition, but a different candidate is used from the `_indexes` array, *e.g.,* `b2_indexes[0]` is used for the first candidate as the Button with label 2, `b2_indexes[1]` as the second candidate (not shown), etc. For brevity, only one option is shown per if statement. If a candidate satisfies the condition, the candidate variable is set (line 23, 46). If no valid candidate can be found, control breaks out of the do loop (line 25, 48).

— an *if block*, if there is only one candidate, *e.g.,* for matching a Floor pattern element that is reached via the `requests_out` link from a Button. No `_indexes` variable has to be used, because only the condition of the single candidate has to be checked.

In practice however, the nested structure is folded where possible into a smaller number of tests, by combining conditions of multiple pattern elements. This way the cyclomatic complexity, and consequently the state space, is decreased significantly. For the changeToUp rule, the LHS can be folded into a single do loop, and the NAC is folded into a single do loop as well.

The conditions that are used to check whether a candidate satisfies the constraints of a pattern element (line 10-22, 35-45) are the following:

1. they are not *null* (*i.e.,* the match candidate is not -1) and there are still possible candidates (line 10, 12, 14, 19, 35, 40);
2. if applicable, they are not the same as a previously matched item (line 20, 41),
3. if applicable, their dynamic type, represented by the `__subtype` attribute, is correct (line 10, 13, 15-17, 21, 36-38, 42-43) and,
4. if applicable, element conditions that are specified are satisfied (line 11, 18, 22, 39, 44-45).

The order of evaluating the candidates as shown in Figure 2.18 can be recognised in the order of expressions in the conditions.

If a match is found for the LHS (line 7-26, 64-65), and no match is found for a NAC (line 32-55), the right-hand side (RHS) of the rule is applied (line 58), which is generated from the difference between the RHS and the left-hand side of the rule. The rule is flagged successful and is exited (line 59-60). Finally the execution jumps back to the rule schedule, which will decide the next step (line 66).

2.4.5 Compilation of the Property Instance

The property instance is translated to an LTL formula (line 5 in Listing 2.1) and *Promela* code. *Promela* code is necessary to evaluate the formula's propositions which are modelled as structural patterns (line 17 in Listing 2.1). The compilation to an LTL formula is done according to the

```
 1 UPDATESTATE:
 2 skip;
 3 d_step {
 4   f0 = 0; // statically set
 5   P0 = 0;
 6   do
 7   :: ((P0 == 0) && (b1 < 3) && (s.floor_[f0].requests_in[b1] >= 0)
 8      && (s.button_[s.floor_[f0].requests_in[b1]].__subtype == ElevatorButtonType
 9          || s.button_[s.floor_[f0].requests_in[b1]].__subtype == DownButtonType
10          || s.button_[s.floor_[f0].requests_in[b1]].__subtype == UpButtonType)
11      && (r.button_[s.floor_[f0].requests_in[b1]].pressed == 1)) ->
12     P0 = 1;
13     break;
14   :: (b1 >= 3) -> break;
15   :: else -> b1++;
16   od;
17   Q0 = 0;
18   e3 = 0; // via id field
19   if
20   :: ((r.elevator_[e3].doors_open == 1)
21      && (f0 == r.elevator_[e3].currentfloor_out)) ->
22     Q0 = 1;
23   :: else -> skip;
24   fi;
25   (...)
26 }
```

Listing 2.7: The compiled proposition patterns of the property.

formulas of Table 2.1. Additionally, quantification patterns are statically evaluated on the design model and for each match an LTL formula is generated, joined by logical *and* in case of *for all* quantification, and logical *or* in case of *exists* quantification. For example, the reachesFloor property of Figure 2.14 for the design model of Figure 2.9 can be compiled to the formula $\square(\neg P0 \vee \diamond(P0 \wedge \diamond Q0)) \wedge \square(\neg P1 \vee \diamond(P1 \wedge \diamond Q1)) \wedge \square(\neg P2 \vee \diamond(P2 \wedge \diamond Q2))$. The two proposition patterns of Figure 2.14 (*i.e.,* the two rightmost structural patterns), are thus translated to six propositions, as both are applied to each of the three floors. The proposition $P0$ represents "a button is pressed at the first floor", $Q0$ represents "the elevator has opened its doors at the first floor", etc. Note how the resulting formula does not only depend on the property, but also on the modelled system.

The propositions of the property must be evaluated after every rule application of the operational semantics. This is done in the `UPDATE STATE` block (line 17 in Listing 2.1), which is shown in Listing 2.7. The evaluation of propositions `P0` (line 5-16) and `Q0` line (17-24) are shown, and `P1`, `P2`, `Q1`, and `Q2` are not shown but are similar, modulo the different value of `f0` (line 4). Since the code represents matching a pattern, it is similar to the code implementing a rule. The main difference however is that we are only interested in whether a match can be found, while it is not important which match is found. Therefore, the choice of candidates can be deterministic, allowing for more optimal and simpler code that only traverses all candidates (line 6-16). If a match is found, the proposition variable becomes 1, else it is 0.

2.4.6 Compilation and Parsing of the Counterexample

If the transition that is followed in the rule schedule dictates that output must be written, the schedule directs to the `OUTPUT` label in the `PRINT STATE` block (line 16 in Listing 2.1) which is

```
OUTPUT:
skip;
d_step {
  printf("%d: ", timestep);
  if
  :: (rule == movedown_last_) -> printf("MOVEDOWN_LAST              ");
  :: (rule == movedown_) -> printf("MOVEDOWN                   ");
  :: (rule == moveup_last_) -> printf("MOVEUP_LAST                ");
  :: (rule == opendoor_up_) -> printf("OPENDOOR_UP                ");
  :: (rule == closedoor_) -> printf("CLOSEDOOR                  ");
  :: (rule == changetoup_) -> printf("CHANGETOUP                 ");
  :: (rule == changetodown_) -> printf("CHANGETODOWN               ");
  :: (rule == initialize_) -> printf("INITIALIZE                 ");
  :: (rule == environmentstep2_) -> printf("ENVIRONMENTSTEP2           ");
  :: (rule == environmentstep1_) -> printf("ENVIRONMENTSTEP1           ");
  :: (rule == opendoor_down_) -> printf("OPENDOOR_DOWN              ");
  :: (rule == moveup_) -> printf("MOVEUP                     ");
  :: else -> skip;
  fi;
  printf("$atompmId:1.pressed=%d;", r.button_[0].pressed);
  printf("$atompmId:2.pressed=%d;", r.button_[1].pressed);
  printf("$atompmId:3.pressed=%d;", r.button_[2].pressed);
  printf("$atompmId:7.pressed=%d;", r.button_[3].pressed);
  printf("$atompmId:8.pressed=%d;", r.button_[4].pressed);
  printf("$atompmId:9.pressed=%d;", r.button_[5].pressed);
  printf("$atompmId:10.pressed=%d;", r.button_[6].pressed);
  printf("$atompmId:0.doors_open=%d;", r.elevator_[0].doors_open);
  printf("$atompmId:0.going_up=%d;", r.elevator_[0].going_up);
  if
  :: (r.elevator_[0].currentfloor_out == 0) ->
    printf("$atompmId:0.currentfloor_out=$atompmId:4;");
  :: (r.elevator_[0].currentfloor_out == 1) ->
    printf("$atompmId:0.currentfloor_out=$atompmId:5;");
  :: (r.elevator_[0].currentfloor_out == 2) ->
    printf("$atompmId:0.currentfloor_out=$atompmId:6;");
  fi;
  printf("\n");
  timestep = timestep + 1;
}
```

Listing 2.8: The code for printing the current state.

shown in Listing 2.8. The last executed rule and state of the system are printed out using the id of the corresponding model element. This information can be used to trace back the value to the run-time model, and allows for playing out a trace. The `printf` statements are only executed during simulation in *Spin*, not during model checking.

After the *Promela* model is generated (step 1 in Figure 2.3), *Spin* uses model checking to find a counterexample of the property (step 2 in Figure 2.3). If a counterexample is found, a *trail* file is generated, which can be simulated in *Spin*, to execute the print state code of Listing 2.8 (step 3 in Figure 2.3). The printed text is directed to a text file, which can be interpreted by a generic parser that creates a trace model (step 4 in Figure 2.3). The trace model can be played out on the run-time model, so that the counterexample is visualised step by step (step 5 in Figure 2.3). Because the trace model usually has this purpose, it is parsed implicitly to memory instead of constructing a E_t instance.

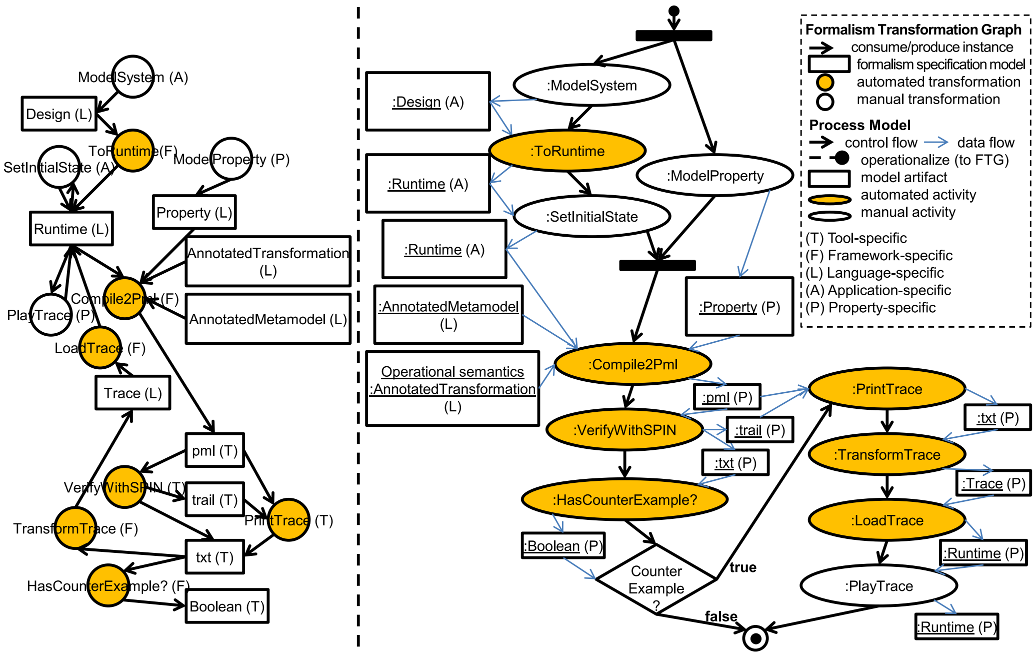


Figure 2.19: FTG (left) and PM (right) of the domain user's activities using *ProMoBox*.

2.5 The Elevator *ProMoBox* in Action

We implemented the *ProMoBox* framework in *AToMPM* [194], and the compiler for models to *Promela* and the parser for text to models were written in Python, using *AToMPM* bindings. Figure 2.19 shows the FTG+PM of the full process of using *ProMoBox* for modelling and verification, including the modelling of the system and property. In accordance with the two DSM phases explained in Section 1.2.1, this modelling process is typically carried out by the domain user and starts where the process of the language engineer of Figure 2.4 stops. The user starts by modelling a system in the design language. The design model can be automatically converted to a run-time model using the generic ToRuntime transformation. This transformation is similar to the migration described in Section 2.3.5, meaning that it only consists of an namespace switch, if necessary. In this case however, obligatory run-time attributes (such as the pressed, going_up and doors_open attributes) and run-time associations with a minimum cardinality greater than 0 (such as the currentfloor association) are required in Elevator and Button instances in the run-time language. Consequently, the result of the transformation is a model conform a "relaxed" run-time language (much like the intermediate metamodel described in Chapter 3), *i.e.,* the run-time language modulo constraints on the minimum cardinality of run-time associations. In this relaxed run-time language, the initial state can be modelled in the SetInitialState activity. In case of the elevator example, the currentfloor link should be set and initial values for attributes should be given. Instead of modelling by hand, default values can be used, if available. In this particular example however, which initial state is chosen is not relevant, as all states turn out to be reachable from one another. In parallel with modelling the system, a property is modelled in the ModelProperty activity.

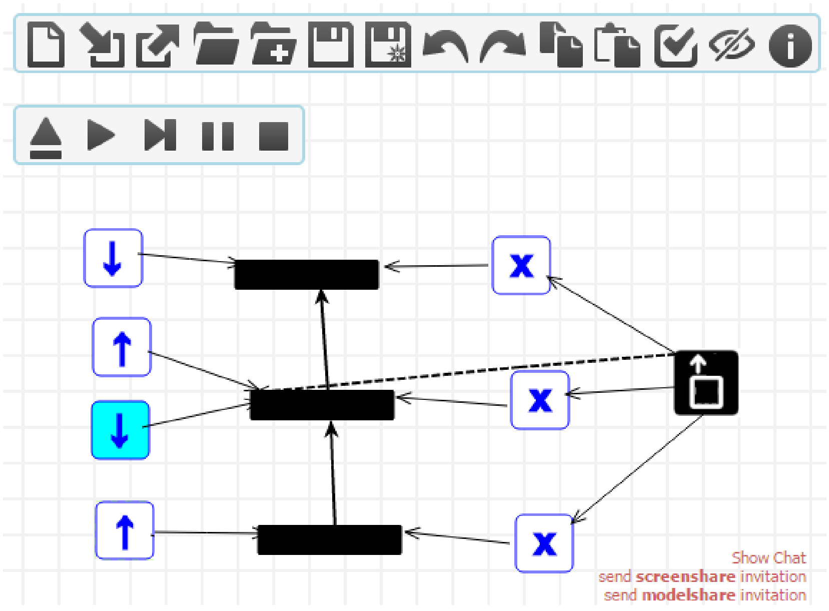


Figure 2.20: Using the toolbar for playing out the counterexample trace in *AToMPM*.

The remainder of the process model in Figure 2.19 explains the five steps of Figure 2.3 in detail. The property model, the run-time model, and the existing annotated metamodel and operational semantics are translated to a *Promela* model as explained in Section 2.4. This *Promela* model, containing an LTL formula, is fed to the *Spin* model checker in the VerifyWithSPIN activity. Depending of the type of the property (liveness or safety), *ProMoBox* automatically instructs *Spin* whether to look for acceptance cycles or not. The report of the verification is stored as a text file. In the HasCounterExample? transformation, the report is automatically analysed to conclude whether a counterexample is found. If none is found, the model checking process finishes and it is concluded that the system satisfies the property. In case of a counterexample, *Spin* produces a `trail` file, containing a counterexample scenario. Subsequently, *Spin* is executed to perform a guided simulation in the PrintTrace transformation, following the scenario described in the `trail` file. In guided simulation, the generated print statements (see Section 2.4.6) are executed, which results in a textual execution trace. The resulting output stream is directed to a text file which is a sequence of system states and transitions. This text file can be automatically transformed to a trace model by a generic parser with the TransformTrace transformation. Note however that for performance reasons, the result of this transformation is an implicit trace model, as actual trace models tend to be rather large – in the order of magnitude of the number of elements in the design model times the number of states in the trace trace. In order to inspect the counterexample at the level of the DSML, the trace model can be automatically loaded in *AToMPM* as shown in Figure 2.20, showing the initial state and a toolbar (shown below *AToMPM*'s main toolbar) to step through the counterexample. A counterexample can be loaded, it can be fully played out, one step can be taken, and a full play-out can be paused or stopped, which is done in the manual PlayTrace activity.

When applying this process to the running example, (maybe unexpectedly) a counterexample

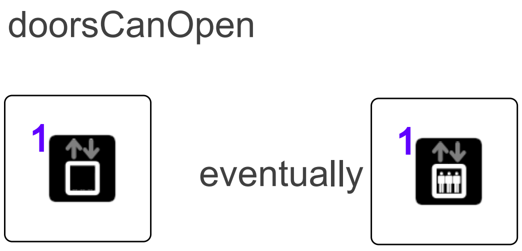


Figure 2.21: The doorsCanOpen property.

for the reachesFloor property is found. When playing out the counterexample as shown in Figure 2.20, it can be seen that the system might get in an infinite loop consisting of three steps: (1) a button is pressed on the floor where the closed elevator currently is, (2) the elevator opens its doors and unlights the button, and (3) the elevator closes its doors. These three steps can be repeated infinitely many times (which is shown when playing out the counterexample), even if buttons on other floors are pressed, thus resulting in an acceptance cycle representing this fairness problem.

We checked the property with *Spin* [89] version 6.4.4 on a 64-bit Windows 7 SP1 PC with an Intel(R) Core(TM) i7 Q 720 CPU at 1.60 GHz (up to 2.80 GHz) with 8 GB of 1600 MHz DDR3 memory. As is typical for *ProMoBox* in case of a counterexample [136, 138, 49], the execution time is short. In this case, *Spin* takes less than 1 millisecond to find a counterexample, and uses 5 KB of memory. Another property is shown in Figure 2.21 stating that if the elevator doors are closed, they will eventually open again. This property does not have a counterexample. Model checking with *Spin* results in a traversal of the entire state space, and takes 48 milliseconds, using 409 KB of memory.

2.6 Evaluation of *ProMoBox*

The *ProMoBox* approach is evaluated in this section.

First, we discuss the performance of the generated *Promela* model. The performance of explicit-state model checkers such as *Spin* is a well-known issue which is inherent to the approach, due to state space explosion [11]. Because the *ProMoBox* approach uses model checking as its verification backbone, it is inherently subjected to these performance issues, which are considered to be beyond the scope of this chapter. However, we present anecdotal evidence that the generated code of the Elevator *ProMoBox* is at least competitive with a manually created *Promela* model.

The contribution of the *ProMoBox* framework is to provide an automated and flexible approach for modelling and verifying properties. Therefore, the main part of this section evaluates these promises, which reflect the MPM principles. More specifically, the level of automation is evaluated by discussing the extent of user input, and the flexibility is evaluated by assessing the impact of evolution of all of *ProMoBox*'s related models.

2.6.1 Model Checking Performance

In this section, we compare the *Promela* model that is generated by *ProMoBox* with a manually constructed model.

The manually created model is a variant of a similar model that is presented in Merz and Navet's "on the verification of real-time systems" [134]. The model presented in the book is simpler than our case study. Consequently we adapted it so that it behaves the same as our rule-based variant. Since it is derived from a model presented by experts in the domain of verification, this model is representative for a model built by an average *Promela* user. The model is shown in Appendix A and corresponds to the running example, which has three floors and seven buttons. In this case however, no buttons are pressed in the initial state, but a simple environment is modelled where buttons can be pressed. We use a meaningless LTL formula $\Box(True)$ that forces a full state space traversal. Note that a checking a different LTL formula yields different execution times and memory consumption, as usual in *Spin*.

The *ProMoBox Promela* model used as running example throughout this chapter, has some slight differences compared to the compiled *Elevator* case of Section 1.3:

— we use the more optimal rule schedule where the change direction rules are evaluated before the move rules, so that the NAC in the change direction rules can be removed as mentioned in Section 1.3;

— similar to the manual *Promela* model, no buttons are pressed in the initial state;

— we use the same LTL formula $\Box(True)$.

Variants of the model in Appendix A and of the *ProMoBox* model are used that have different numbers of floors and buttons. For every number of floors, we use at least as many buttons as the number of floors, so that every floor can be requested, and at most as many buttons as needed to have one elevator button, up button and down button for each floor, with no down button for the ground floor and no up button for the top floor. In the variants, buttons are equally distributed over floors. We cover all variants up to the number of floors that causes *Spin* to run out of memory.

The results are shown in Figure 2.22 and Figure 2.23. Both plots use a logarithmic scale on the vertical axis. All test models are shown on the horizontal axis, indexed by two numbers: the number of floors and the number of buttons. We verified each *Promela* model 50 times, and excluded the 5 highest and 5 lowest outliers. The averages of the execution times without outliers is shown in Figure 2.22. Figure 2.23 shows memory-related resources. The number of states and number of transitions reflect the size of the state space. The state vector size is shown and the memory usage for the state space is shown (excluding memory used for optimisations, which is determined by *Spin* run-time parameters). Note how the required resources grow exponentially with the size of the environment (*i.e.,* the number of buttons) as discussed in Section 2.3.5. The *ProMoBox* models need fewer resources for all test cases.

Figure 2.24 shows the improvement factor of all resources for every test case. The plot indicates no decline in improvement for larger models. Interestingly, the improvement (especially in time but also in memory) increases for the more complex cases that have a larger number of buttons.

As stated before, this comparison serves as an indicative measurement and is by no means evidence that *ProMoBox* generates optimal *Promela* models. For other examples, the execution time or memory consumption of the model checking algorithm may still be impractically high,

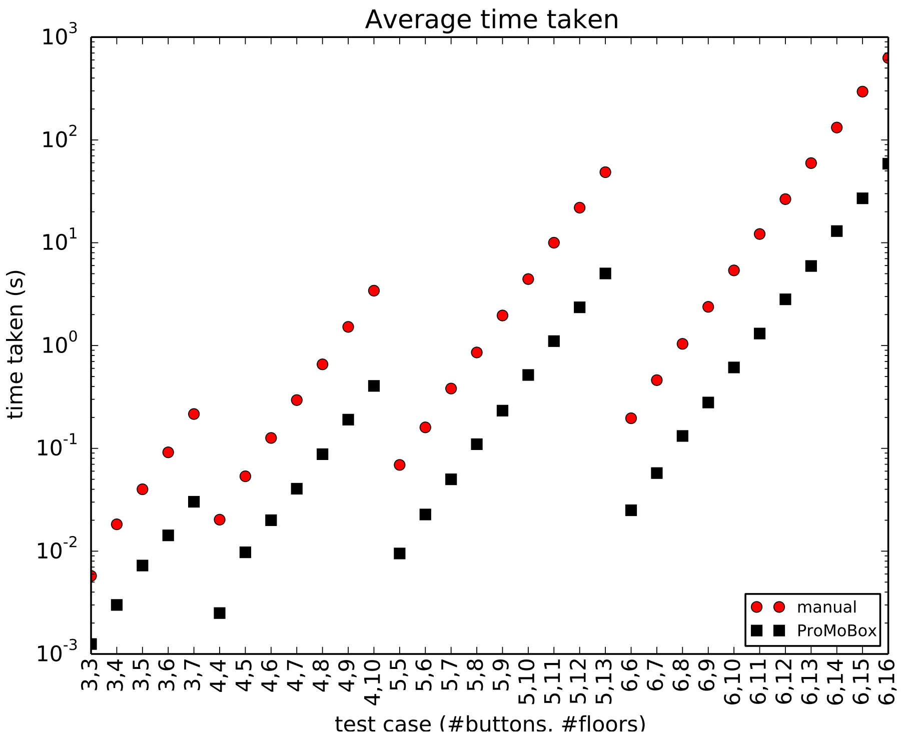


Figure 2.22: Comparison of the state space traversal time.

even after simplification (Section 2.3.5). Although desirable, a full state space exploration may not always be necessary to find counterexamples. According to the “small scope hypothesis”, an essential factor in the applicability of the Alloy Analyzer [91], the majority of errors can be found by testing the model and all of its inputs within a small scope. The breadth-first exploration of *Spin* follows the same principle. In case of *ProMoBox*, we systematically found counterexamples within one millisecond [136, 49], while traversing the full state space took significantly longer and required significantly more memory.

The goal of this comparison is to provide anecdotal evidence that generated *Promela* models are usable for model checking. Further evaluation on the performance of generated *Promela* models diverges from the contribution of this chapter for two reasons. First, the mapping to *Promela* and LTL is only one example of a possible verification backbone for *ProMoBox*. In this respect, a mapping to *Groove* and CTL is shown in Section 2.6.3. Second, optimising the generated *Promela* models requires profound knowledge of *Promela* and *Spin* rather than of the DSML and is thus beyond the scope of this DSM-centred contribution.

2.6.2 Modelling Performance

One of the design objectives of *ProMoBox* is automation. The goal of the approach is to support modelling and verification of properties for DSMLs with a minimal burden for both the language

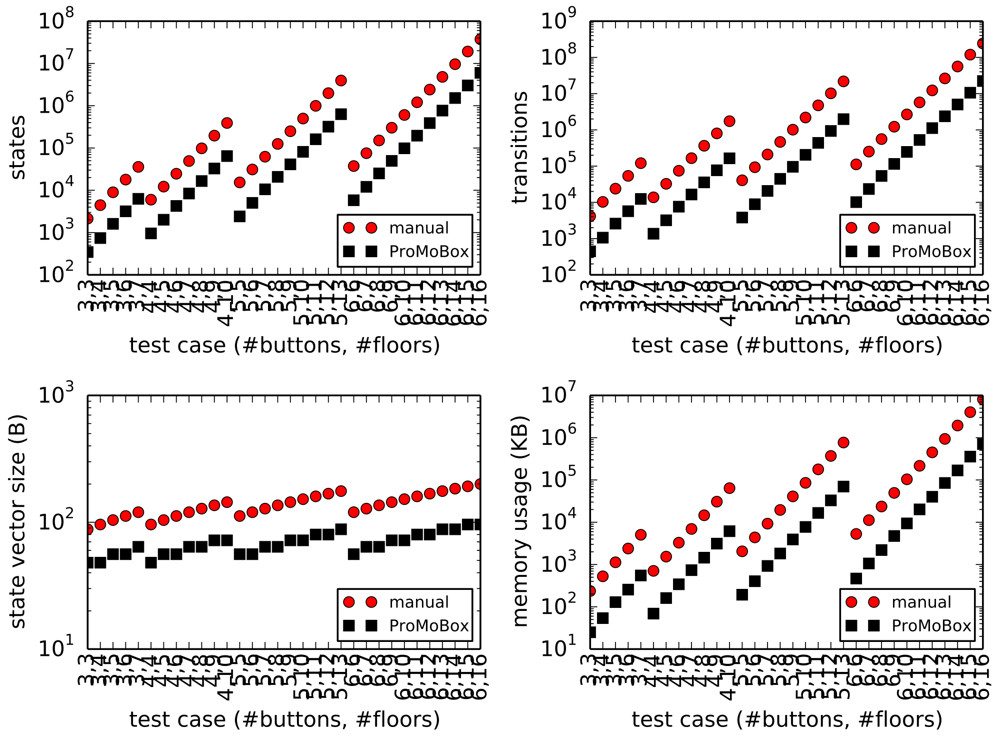


Figure 2.23: Comparison of the state space traversal memory usage.

engineer and the domain expert (who design the DSML) and the domain user (who uses the DSML). In order to assess the contribution with respect to automation, we compare the amount of manual work when using *ProMoBox* with:

— the amount of manual work when using traditional DSM for modelling and verifying properties. In this comparison, the emphasis is on the amount of manual work that is avoided by *ProMoBox*;

— the amount of additional manual work when using traditional DSM without verification. In this comparison, the emphasis is on the amount of manual work required to support property verification using *ProMoBox*.

Comparison with Traditional DSM Including Verification Support

In the literature, we distinguish three approaches to support verification (see also Section 2.7):

— ***approach 1***: no DSML for properties is available, but properties are directly modelled in logic (*e.g.,* [171], following the architecture shown in Figure 2.2);

— ***approach 2***: a DSML for properties is created including a mapping to a verification backbone, but no counterexample parsing is supported (*e.g.,* [100]);

— ***approach 3***: a DSML for properties is created including a mapping to and counterexample parsing from a verification backbone (*e.g.,* [185]).

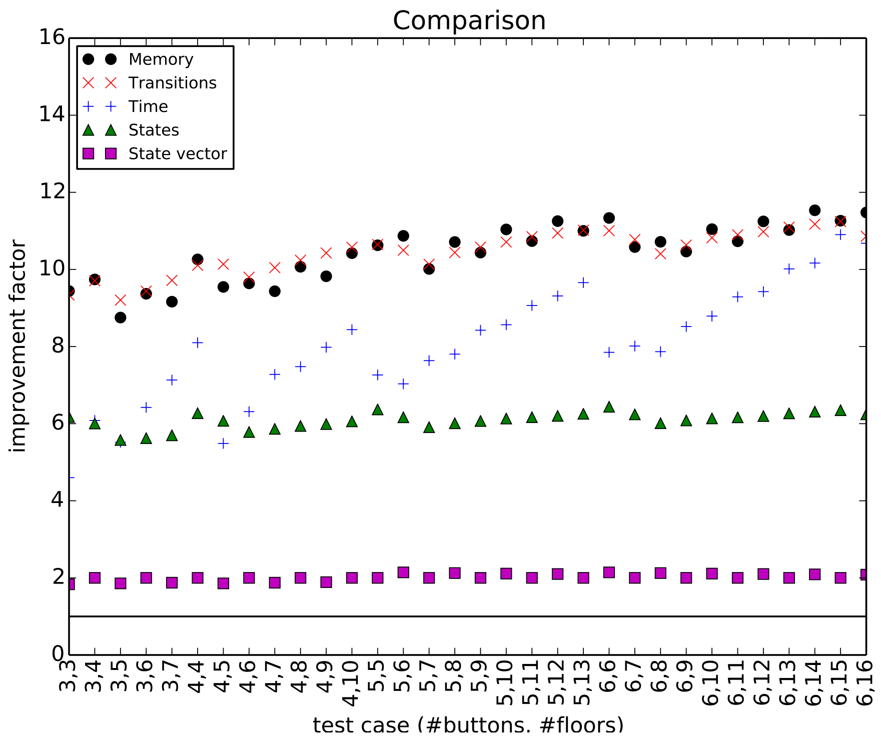


Figure 2.24: Improvement factor of the traversal time and memory usage.

	Appr. 1	*Appr. 2*	*Appr. 3*	*ProMoBox*
Property language modelling	none	manual	manual	generated
System model mapping to verification backbone	manual	manual	manual	automated
Property mapping to verification backbone	none	manual	manual	automated
Counterexample parsing	none	none	manual	automated

Table 2.2: Comparison of the degree of automation between the state of the art and *ProMoBox*.

For this comparison, we only consider approaches that are applicable to DSMLs in general, thus excluding modelling languages that are expressive enough to support property specification and generation (*e.g.*, [128]).

The comparison of existing approaches with *ProMoBox* is shown in Table 2.2. The approaches differ in whether they support, either manually or automatically, the given four language engineering activities. If the activity is not supported (entry "none"), a burden is put on the domain user, because the activity is not raised to the domain-specific level. If the activity is marked "manual", it requires a manual effort from the language engineer resulting in a DSM solution for the domain user. A "generated" marking means that a generative process enables the execution of the activity and “automated” refers to a generic process.

It can be concluded that compared to traditional approaches, the language engineer has less work when using the *ProMoBox* approach for each of the activities, and better support for the domain user is provided compared to the first two approaches.

Comparison with Traditional DSM Excluding Verification Support

In this section we assess how much manual work is needed to extend a DMSL with verification support using the *ProMoBox* approach. Since this comparison reflects the modelling processes, the FTG+PM is analysed to evaluate the level of automation. The *ProMoBox*’s FTG+PM shown in Figure 2.4, Figure 2.7 and Figure 2.19 is compared to the FTG+PM of traditional DSM (Figure 1.2). There are two additional manual tasks for the language engineer, denoted as manual activities with the (L) mark (language-specific): (1) she/he has to annotate the metamodel, and (2) she/he has to annotate the operational semantics transformation model. Since this annotation process requires minimal effort, the additional effort required for the language engineer is limited. For the domain user, relevant activities are marked with (A) or (P) (application-specific and property-specific). She/he has to model her/his property and in case of a counterexample, play it out. Note that the SetInitialState activity is part of the CreateInstance activity in traditional DSM. The two extra activities can be performed at the domain level. In conclusion, the domain user only has to limit his- or herself to the bare necessities of the task at hand, using the most suitable language.

2.6.3 Evolution Performance

The flexibility of *ProMoBox* is evaluated by assessing the impact of evolution of its associated components. A generic approach for evolution of artifacts in a relational ecosystem is presented in Chapter 3. This is done by inspecting the FTG+PM of Figure 2.4, Figure 2.7 and Figure 2.19 once again. We start from the assumption that any model artifact in the process model can evolve. Its (worst case) impact can be directly calculated by transitively following the activities that have the model as input. For instance, a change in the property instance :Property requires a restart of the full verification process. This means that the *Promela* model needs to be compiled again, and the verification needs to be to be executed again, including the transformation and loading of the trace in case of a counterexample and playing out the counterexample:
cost(evolve(:Property)) = cost(Compile2Pml + VerifyWithSPIN + HasCounterExample? + PrintTrace + TransformTrace + LoadTrace + PlayTrace).
The cost of VerifyWithSPIN was illustrated in Section 2.6.1 and is inherent to model checking. The cost of all other automated activities is at most polynomial in the size of the model, and is therefore negligible in comparison with the cost of manual effort. The only manual step required is the PlayTrace activity, and it represents therefore the significant cost of changing :Property. This user cost is limited by keeping this activity to its essence, by providing a domain-specific visualisation and a toolbar. Changing :Runtime has a similar impact, as well as :Design, but for the latter the :SetInitialState activity has to be performed again.

Three other classes of evolution scenarios are distinguished, which cover all possible evolution scenarios in *ProMoBox*. In contrast to the evolution scenario discussed above, these three classes of evolution scenarios all affect artifacts at the language level. We performed each of them on the Elevator *ProMoBox* so that a more precise measure of impact can be presented in addition to simply counting the number of required manual tasks. Evolutions can be conceptually very complicated and can therefore require a tremendous effort, but in this evaluation we assess the *technological impact*, *i.e.,* what the framework's architecture requires the user to do in case of evolution.

Evolution of the DSML

The first scenario is an evolution of the DSML, notably :AnnotatedMetamodel, :ConcreteSyntax and :AnnotatedTransformation in Figure 2.4. Since :AnnotatedMetamodel is created at the very start of the *ProMoBox* process, all subsequent activities have to be redone. There are however shortcuts possible:

— if only :ConcreteSyntax evolves, :AnnotatedMetamodel and the metamodel of the five sublanguages can remain untouched. Only the concrete syntax models have to be regenerated (the automated :FilterCS, :MergeCS and :RAMify activities) in Figure 2.7. In *AToMPM*, instance models like in Figure 2.19 and operational semantics transformation models that use the concrete syntax do not need to change. We validated this by changing the button icons (compared to [136]), which was achieved in a matter of minutes;

— if only :AnnotatedTransformation evolves, only the compilation to the verification backbone and the verification itself have to be redone, similar to changing :Property. No models need to be adapted. We validated this by optimising our operational semantics as explained in Section 2.6.1, by reordering rules and removing the NACs, which was also achieved within minutes;

— in case of an evolution of :AnnotatedMetamodel, it might very well be possible that the changes are non-breaking for one or more of the five sublanguages. For example, if an association annotated with *rt* is added, the design metamodel and input metamodel will remain the same. Consequently, their instance models do not need to be migrated either. In case of a change in a sublanguage, it needs to be regenerated. The change of the sublanguage might be non-breaking (*e.g.,* a non-obligatory association is added), which means that no change in its instances is necessary. In general, co-evolution rules as described in Chapter 3 apply to its instances, which means that an instance might not have to be migrated (in case of a non-breaking change), can be automatically migrated (in case of a breaking and resolvable change), or should be migrated manually (in case of a breaking and unresolvable change). If an instance that changes is input to the Compile2Pml activity, the full verification process as described above has to be restarted. Incremental techniques might be used to only partly regenerate or recompile, but since the impact of the automated activities can be neglected, this is left as future work.

In [49] we went a step further by applying *ProMoBox* to a DSML for the gestural interaction domain, including a simplification step as described in Section 2.3.5. This process only took slightly longer than a traditional DSM process. This confirms the discussed of Section 2.6.2.

Evolution of the Property Language

The second scenario is an evolution of the property language, by evolving the metamodel template for properties in Figure 2.7 (as shown in Figure 2.13). As indicated by the (F) mark of the template, this is in fact an evolution of the *ProMoBox* framework itself, instead of an evolution of its input

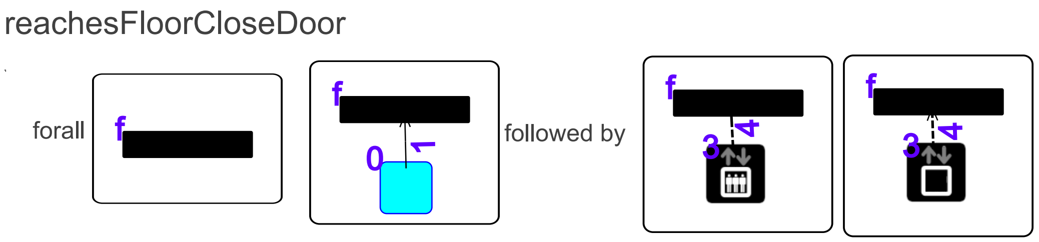


Figure 2.25: The variant of the reachesFloor property: after a button is pressed, the elevator will eventually open its doors at the corresponding floor, and then close them again.

models. A consequence of the evolution is that the concrete syntax template might have to be updated and that the property language has to be regenerated. Also, the co-evolution rules apply to its instances, possibly requiring the full verification process as described above to be redone. In this case, since the template has changed, the Compile2Pml activity needs to be adapted as well, so that the changed language is adequately compiled.

We added an additional pattern called “chained response” [55] to the property language template, which is similar to “response” but has two rather than one effects that need to happen in the given order. Using this new pattern we were able to extend the reachesFloor property by adding a second restriction that the doors should close again after they were opened. This new property is shown in Figure 2.25. Implementing this pattern in the template and adapting the compiler took less than one hour, and no existing property models had to be changed, since the evolution can be classified as an additive, non-breaking change.

The evolution of the following models has a similar impact:

— the impact of changing the template of another sublanguage is similar. In particular, changing the trace language template differs slightly as it requires the parser TransformTrace to be changed instead of the compiler Compile2Pml;

— the impact of changing the concrete syntax of a template is fully automated. It only requires the regeneration of the concrete syntax of the sublanguages;

— the impact of changing the annotations model of Figure 2.5 is fully automated with respect to the language engineer. It requires all sublanguages to be regenerated. For the sublanguage instances co-evolution rules apply, so depending on the change the instance might have to be migrated and the verification process might have to be redone.

One might wish to add *e.g.,* real time to the property language. We consider such changes too intricate to be called an evolution, because they would require not only the property language to change, but also the input and trace language, and even the transformation language for operational semantics. This is interesting future work however.

Evolution of the Verification Backbone

A last type of change that should be investigated is replacing the entire evolution backbone. To illustrate this, we replaced the mapping to *Promela* and LTL by a mapping to *Groove* and CTL. Since models in *Groove* are graphs and transformations are rule-based just like in our view on DSM, the mapping is more straightforward than to *Promela*. The downside is that *Groove* is not as

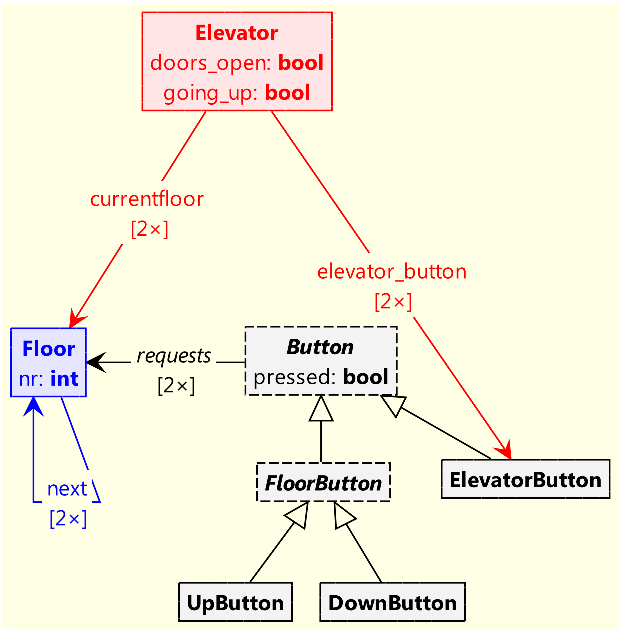


Figure 2.26: The *Groove* type graph generated from the metamodel.

expressive as *Promela* (notably the lack of NAC patterns and a limited scheduling language) and the performance of model checking is significantly lower. Regarding the mapping to CTL, similar to the LTL formulas presented in Table 2.1, Dwyer et al. [55] also provide a mapping to CTL for every temporal pattern.

The mapping to *Groove* took about four days to implement, including getting familiar with *Groove*. A similar compilation strategy is followed as shown in Figure 2.17, where AST_{PS} is now *Groove*-specific, and the target is divided in multiple different models implementing *Groove* type graphs, instances, rules, environment and property propositions (as XML-files), and a rule schedule and CTL formula (in textual syntax).

The run-time metamodel of Figure 2.10 is translated to a *Groove* type graph as shown in Figure 2.26. Colors are added for clarity, and the multiplicities are encoded in the edges (not visible).

The *Groove* equivalent of the instance depicted in Figure 1.6 is shown in Figure 2.27. A generic concrete syntax is used, with rectangles, arrows with labels, and expressions.

The translation of the MoveUp rule (see Figure 1.7) is shown in Figure 2.28. The pattern includes the LHS and RHS, where all pattern conditions represent the LHS, and all statements, creations and deletions represent the RHS and are applied if a match is found. In this case, the RHS consists of deleting a currentfloor link (dashed) and creating a new one on the floor above (bold). This notation allows for concise rules. The left part of the rule visualises a simple comparison operation representing the LHS condition.

The complete rule schedule is shown in Listing 2.9. This schedule is generated from the more optimised schedule that first evaluates the move rules, followed by the change direction rules. The main control structure of the schedule is an infinite while loop, representing the simulation loop.

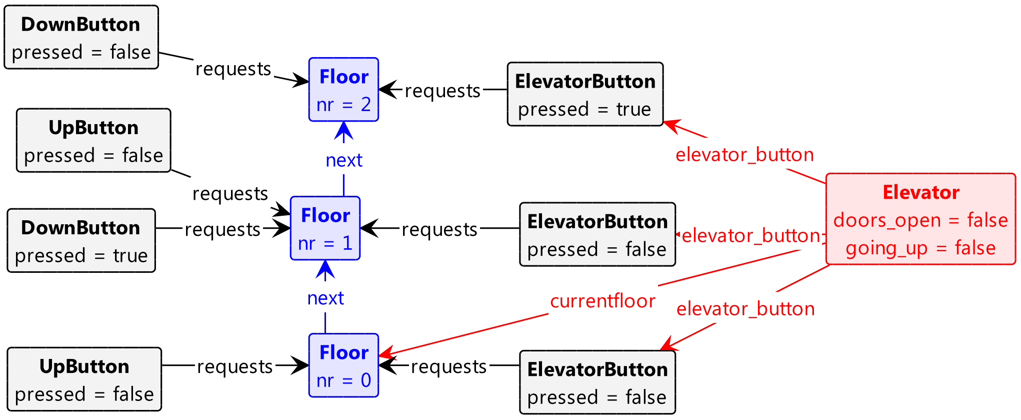


Figure 2.27: The *Groove* instance graph generated from the run-time model.

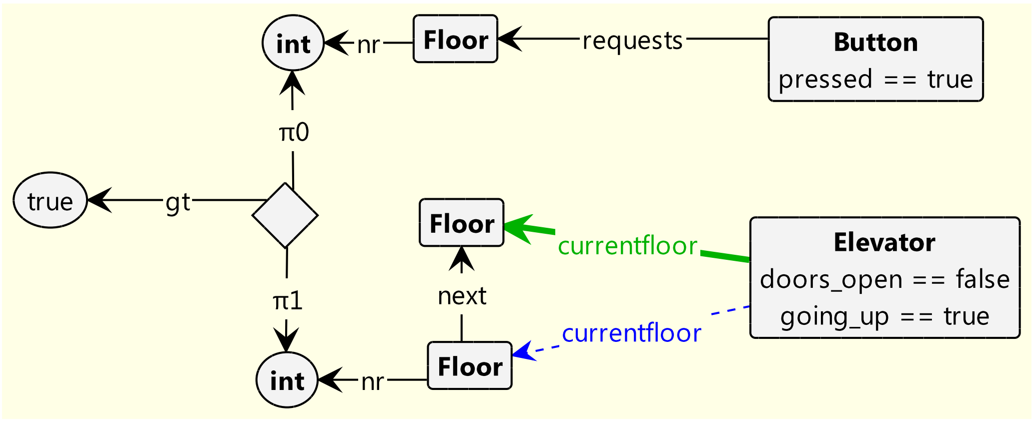


Figure 2.28: The *Groove* rule for MoveUp.

The rules are tried in the correct order. If successful, the environment is evaluated if necessary, and a new iteration of the main while loop is started. If unsuccessful, the next rule is tried. Note that, since *Groove* does not support conditions on pattern element types, the openDoor_up and openDoor_down rules have to be split into two rules: one matching a directional button and one matching an elevator button.

The environment distilled from the annotated metamodel is shown on the left of Figure 2.29, and presses a button that is not yet pressed. Additional to this rule, an empty rule nothing is created to give *Groove* the option to not perform an environment step.

According to Dwyer's specification patterns [55] and taking quantification into account as presented in Section 2.4.5, the property is translated to the CTL formula:

```
A[!Q0 W (Q0 & AF(P0))] & A[!Q1 W (Q1 & AF(P1))] & A[!Q2 W (Q2 & AF(P2))].
```

The propositions are translated to *conditions* in *Groove*, which are rules that do not have a side effect. *Groove* automatically evaluates after every state change whether or not a match for these conditions can be found. The condition `P0` is shown on the right in Figure 2.29.

```
while(true) {
  try {
    openDoor_up_UpButton | openDoor_up_ElevatorButton;
    Environment | nothing;
  } else {
    try {
      openDoor_down_DownButton | openDoor_down_ElevatorButton;
      Environment | nothing;
    } else {
      try {
        closeDoor;
        Environment | nothing;
      } else {
        try {
          moveUp;
          Environment | nothing;
        } else {
          try {
            moveUp_last;
            Environment | nothing;
          } else {
            try {
              moveDown;
              Environment | nothing;
            } else {
              try {
                moveDown_last;
                Environment | nothing;
              } else {
                try {
                  changeToDown;
                } else {
                  try {
                    changeToUp;
                  } else {
                    Environment;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 2.9: The rule schedule in *Groove*.

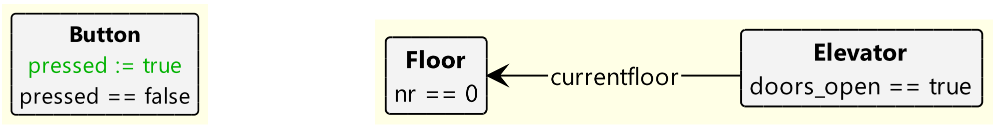


Figure 2.29: The *Groove* environment generated from the annotated metamodel (left) and the *Groove* proposition rule for $P0$, the elevator opens its doors at the ground floor (right).

2.6.4 Assumptions and Limitations

To conclude the evaluation of *ProMoBox*, we discuss the assumptions and limitations of the approach.

Format of the DSML

It is assumed that we can express the abstract syntax of the DSML as a metamodel in the form of a class diagram. The concrete syntax is defined graphically by icons for every abstract syntax concept. The semantics is given by a transformation model with a rule schedule supporting control flow and graph transformation rules. Under these conditions, an operational *ProMoBox* can be generated from any DSML.

Boundedness

The rule-based nature of the operational semantics ensure a step-wise, state-based semantics. In its current state, *ProMoBox* supports DSMLs that have a notion of state, so that temporal properties can be checked. Since we apply explicit state model checking, the number of possible states must be bounded. This is guaranteed by limiting the multiplicity of the run-time elements. If such boundedness is not achieved in the metamodel because of an infinite multiplicity value, this value must be bounded (possibly through abstraction) in order to allow model checking. Likewise, other simplification steps might be performed to ensure boundedness, as explained in Section 2.3.5.

Format of the Properties

We currently support temporal properties with quantification and structural patterns. The properties can be mapped to LTL and CTL, so the approach can be considered representative for a wide range of properties. *ProMoBox* does not include support for the “next”-operator (*i.e.,* something must happen immediately after something else), as “immediately” can be interpreted in many ways. A promising direction is to use the principle of the conceptual time step (see Section 2.3.4) and define “immediately” as “in the next conceptual time step”.

Because of *ProMoBox*’s flexibility, we feel that the approach described in this chapter can be reused for different kinds of properties by defining generic mappers to tools supporting model checking with *e.g.,* OCL, real time properties, or properties using distributions. The target tool has to be expressive enough so that a correct structure and operational semantics can be defined, *i.e.,* all elements can be queried, variables can be stored and updated throughout the evaluation of the temporal formula (context-dependency), etc. The key to automation of the approach is that it is defined at the meta-level (class diagrams, concrete syntax definitions, and rule-based transformation with scheduling), in combination with predefined, generic templates.

We use a combination of natural language and patterns as concrete syntax for properties. Although very expressive, simple properties can be cumbersome to model and can still be confusing, which was the very problem *ProMoBox* tried to solve. The contribution of *ProMoBox* is however to provide means to ease the specification and verification of properties. We showed that because of the template-based approach the modeller can easily change the concrete syntax, if for example, a user prefers a visual syntax for patterns. Also, syntactic sugar may be added to easily express and visualise features that occur often. For example, the pattern structure might be bypassed if it consists of a single element. In that respect, and because a domain-specific concrete syntax is used, we feel that the *ProMoBox* framework adequately addresses the understandability of the properties.

Scalability

As scalability limitations are inherent to model checking, it remains the main concern. The compiler generating *Promela* could be further optimised. On the one hand, further generic optimisations can be applied to the compiler by a cooperation between a DSM and *Promela* expert. On the other hand, the compiler might be extended to take user-defined optimisation information into account. For example, since pattern matching is the bottleneck of rule-based transformation, search plans [208] can be incorporated in the approach, to allow an optimal matching order of pattern elements.

Another way to contain the scalability issue is to extend and quantify the modelling guidelines, so that a prediction of the model checking time and memory consumption can be given. To this end, extensive empirical research is needed to quantify the relationship between model characteristics and model checking performance. This relationship is different for every verification backbone.

A radically different solution to the problem of scalability would be to not map to a model checking approach, but instead use test case generation techniques to generate relevant test cases in the form of input models and trace models (oracles). Tests are executed by using the input models as initial state, applying the operational semantics transformation, and comparing (by using model comparison, *e.g.,* the DSMDiff algorithm [120]) the resulting trace with the oracle. This illustrates how *ProMoBox* benefits from its flexible modelling approach, because mappings to different semantic domains can be implemented. However, this research direction is not yet investigated for the *ProMoBox* approach. In Section 2.8 this thread of future work is discussed more.

2.7 Related Work

With respect to the contribution of this chapter, we distinguish two types of related work. First, we consider approaches that translate models to formal representations to specify and verify properties that are created specifically for one modelling language. Second, we discuss approaches that have a more general view on providing specification and verification support for different modelling languages.

2.7.1 Specific Solutions

In the last decade, a plethora of language-specific approaches have been presented to define properties and verification results for different kinds of design-oriented languages. For instance, Cimatti et al. [32] have proposed to verify component-based systems by using scenarios specified as Message Sequence Charts (MSCs). Li et al. [118] also apply MSCs for specifying scenarios for verifying concurrent systems. The CHARMY approach [158] offers amongst other features, verification support for architectural models described in UML. Collaboration and sequence diagrams have been applied to check the behaviour of systems described in terms of state machines [26, 104, 103]. These mentioned approaches are just a few examples that aim at specifying temporal properties for models and verifying them by model checkers (see [66] for a survey). They have in common that they offer language-specific property languages. However, these approaches are not aiming to support language engineers in the task of building domain-specific property languages.

TimeLine [185] is specifically designed to address the design of temporal properties in a visual manner. Analogous to LTL formulas, these TimeLine properties can be transformed to *Promela never claim*s (*i.e.,* Büchi automata), which can be directly used for model checking by *Spin*. TimeLine was used to convert informal requirements written in English to formal requirements that are still very readable. The problem that TimeLine addresses is highly related to the problem statement of this chapter, but in our approach we generalised this approach for DSM.

2.7.2 Generic Solutions

In [207], Varró presents an approach in which a metamodel with operational semantics and an instance model can be transformed to a transition system. Transition systems are used as the *de facto* interchange standard of the Symbolic Analysis Laboratory [16], a framework for combining tools for formal methods. Safety properties and deadlock can be analysed. The approach also makes the distinction between static and dynamic language concepts for reducing the state space.

Rivera et al. map models and their operational semantics of DSMLs to rewriting logic [173], as well as metamodels [172]. Maude [35] can verify properties using rewriting logic, so operational semantics can be subjected to analyzing methods provided out-of-the-box of Maude environments such as reachability analysis and checking of temporal properties specified in LTL. The approach maps rules to rewriting logic (which is in essence rule-based), while our mapping to *Promela* supports a broader platform for rule-based semantics.

There are some approaches that aim to shift the specification and verification tasks to the model level in a more generalised manner. First of all, there are approaches that propose OCL extensions, often referred to Temporal OCL (TOCL), for defining temporal properties on models [221, 94, 18]. As OCL may be combined with any modelling language, TOCL can be seen as a generic model-based property language as well. In [217, 36] the authors discuss and apply a pattern to extend modelling languages with events, traces, and further run-time concepts to represent the state of a model's execution and to use TOCL for defining properties that are verified by mapping the design models as well as the properties expressed in TOCL to formal domains that provide verification support. In addition, not only the input for model checkers is automatically produced, but also the output, *i.e.,* the verification results, is translated back to the model level. The authors explain the choice of using TOCL to be able to express properties at the domain level, because TOCL is close to OCL and should be therefore familiar to domain engineers. However, they also state that early feedback of applying their approach has shown that TOCL is still not well suited to many domain engineers and they state in future work that more tailored languages may be of help for the domain engineers. The work presented in this chapter goes directly in this direction by enabling domain engineers to use their familiar notation for defining properties and exploring the verification results.

Another approach that aims to define properties on the model level in a generic way is presented by Klein and Giese [100]. The authors extend a language for defining structural patterns based on Story Diagrams [62] to allow for modelling temporal patterns as well. The resulting language allows to define conditionally timed scenarios stating the partial order of structural patterns. The authors argue that their language is more accessible for domain users, because their language allows decomposition of complex temporal properties into smaller ones by if-then-else decomposition and quantification over free variables. Their approach is tailored to engineers that are familiar with

UML class diagrams and UML object diagrams as their notation is heavily based on the concepts of these two languages. Furthermore, they explain how the specification patterns of Dwyer et al. [55] are encoded in their language, but there is no language-inherent support to explicitly apply them. In our work, we tackle these two issues in the context of DSM by reusing the notation of domain users for specifying properties and providing explicit language support for specification patterns.

Finally, da Costa Cavalheiro et al. [41] present specification patterns for describing properties over reachable states of graph grammars. These specification patterns are purely defined on graph structures (*i.e.,* nodes and edges) and thus are reusable for any modelling language. However, the authors do not discuss integration with current modelling languages to use such specification patterns for specific properties.

None of these approaches supports visual, domain-specific syntax for all used models as in our approach. Moreover, no solutions exist that use a generative approach to define sublanguages similar to the used DSML.

2.8 Conclusion and Future Work

In this chapter, we presented a solution in the form of *ProMoBox*, a framework that integrates the definition and verification of temporal properties in discrete-time behavioural DSMLs, whose semantics can be described as a schedule of graph rewrite rules. Thanks to the expressiveness of graph rewriting, this covers a very large class of problems. With *ProMoBox*, the domain user models not only the system with a DSML, but also its properties, input model, run-time state and output trace. The DSML is thus comprised of five sublanguages, which share domain-specific syntax. The sublanguages are generated from a single metamodel to keep them consistent and to avoid duplication, that is annotated to denote the role of each language concept. he operational semantics of the DSML is modelled as a transformation and is annotated with information about input and output. The modelled system and its properties are transformed to *Promela*, and properties are verified with *Spin*, a tool for explicit state model checking In case a counterexample is found, its execution trace is transformed to the domain-specific level as a trace model, which can be played out. Thus, whilst modelling and verifying properties, the domain user is shielded from underlying notations and techniques. Following MPM principles, the process of the *ProMoBox* framework is explicitly modelled in a Formalism Transformation Graph and Process Model. We used an elevator controller as a running example, and showed how the five sublanguages can be generated and how properties can be verified with *ProMoBox*.

We evaluated the *ProMoBox* approach in three ways. Firstly, we assessed the verification time in *Spin* of the generated *Promela* model, and anecdotally showed that execution times and memory consumption are no worse than their hand-made counterparts. Secondly, we showed that *ProMoBox* is a lightweight solution by comparing with existing DSM approaches. Thirdly, we investigated the impact of evolution of models in *ProMoBox*, to evaluate the approach's sensitivity to change. This included replacing the verification backbone of *Promela* and LTL by *Groove* and CTL.

To summarise, we return to the research questions stated in Section 2.1.

— *RQ 1.1.* What languages are needed at the domain-specific level for specifying and verifying

properties?
We identified five sublanguages for modelling design, run-time, input, traces and properties with each their specific task in Section 2.3.1;

— *RQ 1.2.* How can these languages be generated from a specification of a design DSML?
We presented a generation process for each of these languages in Section 2.3.3, based on templates and (meta)model transformation;

— *RQ 1.3.* How can the verification activity, in this case by model checking, that uses these languages, be automated?
We described a transformation to *Spin* and a transformation from text output produced by *Spin* in Section 2.4;

— *RQ 1.4.* What effort is required to enable verification support for a traditional DSM solution?
We described the process of enabling verification support in Section 2.3.5. We assessed the effort in Section 2.6.2 where we concluded that the additional effort is limited to annotating the DSML metamodel and operational semantics;

— *RQ 1.5.* Does the use of the presented approach result in a similar model checking time as its non-domain-specific counterpart?
We anecdotally showed that the model checking time in the *ProMoBox* framework for *Elevator* models is decreased in Section 2.6.1;

— *RQ 1.6.* What is the impact of changes of/to different components of the approach?
We assessed the impact of changing any of the artifacts in the framework in Section 2.6.3 by identifying three classes of evolution scenarios;

— *RQ 1.7.* What are the assumptions and limitations of the approach?
We identified the limitations and assumptions of the *ProMoBox* in Section 2.6.4, describing the requirements the DSML must adhere to, boundedness of the modelled system, the restrictions on the property format, and the scalability of the approach which is inherent to model checking.

In conclusion, *ProMoBox* provides a solution for the specification and verification of properties in a highly flexible and automated way, according to MPM principles.

An interesting thread for future work is checking different kinds of properties, such as real-time properties. In [107, 76] the authors extend Dwyer's specification patterns with real-time information. These properties could be mapped to a tool for verifying real-time systems, such as UPPAAL [14], which allows the user to specify a system as a timed automaton and specify temporal properties with time bounds.

Furthermore, it is interesting to investigate under what circumstances the model checking process does not have to completely re-executed after a change in the design model or property. Incremental model checking is not supported by *Spin*, though recently, an approach has been described by Molnár *et al.* [142]. Incremental approaches can have a similarly beneficial impact in the context of computationally expensive model queries [201].

Moreover, in more complex models, the model checking approach falls short however because of its computational complexity. Therefore we aim to investigate a more scalable alternative to model checking, in the form of test case generation. We believe that the five sublanguages are highly suitable to address this. A test case can be encoded as a run-time model and an input model.

A trace model (or a variation thereof representing a trace pattern rather than a trace) can serve as an oracle. We want to investigate whether the test case and oracle can be generated from a design model and a property model, thus requiring the same input as in the current *ProMoBox* approach. The test case generation strategy can then be plugged in, whether it be random selection according to a normal distribution, search-based testing [129], or a different strategy. Since testing does not guarantee correctness like model checking does, running the test case generation phase should return a coverage report [140]. This coverage report should be presentable at the DSM level as well, and could include covering every transition in the transformation schedule, every path in the transformation schedule, every element in the design model, every element of the input metamodel, etc. We aim for an equally flexible and automated approach. We have done preliminary work on test case generation for *ProMoBox* with Alexandre Petrenko (CRIM, Montréal, Canada) and Manuel Wimmer (TUWien, Vienna, Austria).

CHAPTER 3
Evolution of Domain-Specific Modelling Languages

Abstract. In Domain-Specific Modelling (DSM), evolution is inevitable over the course of the complete life cycle of complex software-intensive systems and more importantly of entire product families. Not only instance models, but also entire modelling languages are subject to change. This is in particular true for domain-specific languages, whose language constructs are tightly coupled to an application domain. The most popular approach to evolution in the modelling domain is a manual process, with tedious and error-prone migration of artefacts such as instance models as a result. This chapter provides a taxonomy for evolution of modelling languages and discusses the different evolution scenarios for various kinds of modelling artefacts, such as instance models, metamodels, and transformation models. Subsequently, the consequences of evolution and the required remedial actions are decomposed into primitive scenarios such that all possible evolutions can be covered exhaustively. These primitives are then used in a high-level framework for the evolution of modelling languages. We suggest that our structured approach enables the design of (semi-)automatic modelling language evolution solutions.

3.1 Introduction

In software engineering, the evolution of software artifacts is ubiquitous [131]. Diverse artifacts such as programs, data, requirements, and documentation may all evolve. In DSM, where modelling languages play a central role, evolution occurs not only at the level of models, but also at the level of modelling languages. This is in contrast with general-purpose programming languages such as C++ where programs evolve during the software development cycle, but not the programming language. Language evolution applies in particular to DSMLs, where relatively frequent changes in the problem domain as well as in the implementation target domain (due to for example external technical or strategic decisions) must be reflected in the respective languages, as shown in an industrial context by Safa [180]. This is to maintain the high coupling between domain and language. The first problem is the need for rapid development techniques for DSMLs, as they are created and modified frequently during the life-cycle of the system they are used for. The second, and far greater problem is that possibly large numbers of modelling artifacts such as instance models or transformation models developed may become invalid and unusable when a related DSML is modified/evolved. Early adopters of DSM dealt with language evolution issues manually [180]. However, this approach, as well as an ad hoc approach to any language change, is tedious and error-prone [197]. The reason for this is that syntax of languages such as UML [153] and BPMN [151], which have evolved considerably over the last few years, easily comprise several hundreds of element types. Also, the semantic differences resulting from this evolution, either intended or intentional, can be subtle. Hence, dealing with evolution requires in-depth knowledge of the language as a whole. Without a proper scientific foundation, as well as methods, techniques and tools to support evolution DSM cannot live up to its promise of ten-fold productivity increase [96]. This becomes apparent when projects span longer periods of time [180]. Since the problem of modelling language evolution was first identified by Sprinkle and Karsai [187], the general problem has only grown in importance, yet still remains largely unsolved. The importance of modelling language evolution is further evidenced by the attention it receives in the research community. Most contributions in this field are focused on (semi-)automatic model differencing [30] and on the co-evolution of instance models [86]. Nevertheless, there are other artifacts that might have to co-evolve. In this context, we define *syntactic evolution* and *semantic evolution*.

Syntactic Evolution

To obtain insight into the consequences of evolution, let us go back to Figure 1.1 and exhaustively explore all possible evolution scenarios. When MM_{Lang} evolves, all instance models m have to co-evolve. As the relations of Figure 1.1 suggest, the evolution of MM_{Lang} might affect other artifacts as well. First, similar to m, the domain and/or image of transformations such as κ_i, π_i, T_j and $[[.]]_D N$ may no longer conform to the new version of the metamodel. As a consequence, these transformations must also co-evolve. This makes all conformance relations valid once again, which means that the system is syntactically *consistent* again. *Consistency* pertains to a collection of models of a system at a *single point in time*. Metamodel evolution leads to the need for model instances and related transformation models to co-evolve.

However, there are more scenarios. Firstly, it is possible that the metamodel changes in such a way that the co-evolved models become structurally different. This ultimately means that each transformation defined for each co-evolved model has to be re-executed when the related models are needed. The resulting co-evolved models can also be structurally different, so a chain of evolution transformations may be required.

Secondly, changes made to one metamodel can have repercussions on metamodels or other models on the metalevel. For example, when an element is added to a metamodel, a new element is often also added to the concrete syntax model in order to be able to visualise this new construct. Other instances of this scenario can be found in *ProMoBox* (see Chapter 2), where the evolution of the annotated metamodel has to be propagated to the metamodels of all the sublanguages. Thus, there is a need for metamodel co-evolution.

Thirdly, until now, we only discussed metamodel evolution as the driving force behind co-evolution. Evolution of other artifacts, such as instance models and transformation models, should also be taken into account. Normally the case of the evolution of a model m does not enforce a change of a metamodel: related models such as $T_j(m)$ can co-evolve by executing the appropriate transformations T_j. Note, however, that a co-evolved model may itself be a metamodel, thereby possibly triggering a cascade of further co-evolutions.

The case of the evolution of a transformation model can become complex. Often, the evolved transformation simply has to be re-executed on each model in this domain. This restricts a transformation evolution to remain compliant with its source and target metamodels, which may not be desired. For example, a new language might be created by mapping rules for each language construct of an existing language. This is particularly convenient for creating a concrete syntax language. The aforementioned complications stem from two special cases of transformation evolution. Firstly, the evolution of either the parsing or rendering function requires the other one to co-evolve in order to maintain a meaningful relation between abstract and concrete syntax. Such a co-evolution can be generalised to any bi-directional transformation. Secondly, the evolution of the semantic mapping function requires a means to reason about semantics, which are discussed in the next section.

Semantic Evolution

As mentioned above, semantics of a model are defined by means of a translational semantics transformation or operational semantics transformation. In case of translational semantics, analyses can be performed on models in this semantic domain (*e.g.,* model-checking a given LTL formula, or finding state invariants in a Petri net) in order to verify whether the model satisfies given properties. Just like a *ProMoBox* mapping to a verification backbone, a semantic mapping function is constructed in such a way that some properties hold both for a model and for its image under the semantic mapping. These common properties have to be maintained throughout evolution. An evolution is a semantic evolution if the set of properties that hold changes. This typically happens when the requirements of a system change.

In general, a model m (whose semantics are given by a semantic mapping function $[[.]]_{TS}$ as shown in Figure 1.1) has properties $P(m)$ that hold that are identical to the properties of that model in its semantic domain $P([[m]]_{TS})$. If a model m with properties $P(m)$ in some formalism evolves to m', then $P(m')$ must be identical to $P(m)$. Of course, when semantic changes are implemented on purpose (these are new requirements of the system, formalised by a change in the set of properties) they have to be taken into account when comparing properties before and after the evolution process. In other words, it must be guaranteed that they are equal modulo the intended changes. A similar rationale holds for operational semantics. When two versions of a system are (a) equal modulo their intended syntactic and semantic changes and (b) syntactically consistent, the evolution of the system is said to be *continuous*. *Continuity* pertains to a collection of models of a system as they evolve *over time*. Only continuous evolutions are deemed useful. In this thesis, the relationship between properties before and after evolution is not further explored. Our approach relies on evolution that can be captured syntactically (*i.e.,* are modelled explicitly), which includes *e.g.,* a transformation model representing the translational or operational semantics. However, since no general way of explicitly modelling (abstraction) relationships between properties exists as of today, a syntactic approach will not be applicable in the context of continuity. The explicit modelling of property relationships is a research topic in its own right [12], and is presented as related and future work in Section 3.4.5 when discussing the automation prerequisites for our framework, and Section 3.6 when discussing future work.

Research Goals

After sketching the phenomenon of modelling language evolution and introducing new terminology, we can reflect on the research objectives of this chapter:

— the *cause* of this research is the observation that modelling languages evolve;

— the *goal* is to maintain consistency and continuity;

— as a consequence, there is a *need* for co-evolution of modelling artifacts;

— the *approach* to achieve co-evolution is to devise *migration* operations.

This chapter presents guidelines to achieve these objectives. The aim is to be *complete*, meaning that all possible scenarios are covered. In order to achieve this, we break down the phenomenon of modelling language evolution, so that every single aspect can be explored exhaustively.

The general research question of this chapter is:

RQ 2. What is a solution according to MPM principles for dealing with the consequences of language evolution?

More specifically, the following research questions are posed:

— *RQ 2.1.* What is the role of syntax and semantics in language evolution?

— *RQ 2.2.* What is a complete set of language evolution scenarios that incorporates all possible evolution scenarios?

— *RQ 2.3.* How can transformation models be co-evolved in case of language evolution?

— *RQ 2.4.* How can we cover every possible case of language evolution?

— *RQ 2.5.* What are the assumptions and limitations of the approach?

The remainder of the chapter is organised as follows: Section 3.2 introduces a way to tackle evolution of modelling languages by de-constructing the problem into primitives. Section 3.3 uses this information to assemble solutions for evolution scenarios. Section 3.4 presents a framework and algorithm for the evolution of modelling artifacts in a DSML relational ecosystem when languages evolve. Section 3.5 discusses related work, and Section 3.6 concludes the chapter and describes future work.

3.2 De-constructing Evolution

The concept of language evolution is heterogeneous in terms of affected modelling artifacts. In order to deal with this problem uniformly, we first look for basic building blocks for evolution. First in Section 3.2.1, we introduce some concepts from the literature to shape the context of language evolution. Then in Section 3.2.2, we look into evolution in the context of the language-instance relationship. In Section 3.2.3 we continue by explaining the projection problem. This turns out to be important when discussing evolution in a full DSML relational ecosystem, which is done in Section 3.2.4 to 3.2.9. This leaves us re-construct evolution and present a solution to transformation co-evolution in Section 3.3.

3.2.1 Metamodel Evolution and Co-Evolution

In order to be able to model evolution in a DSML relational ecosystem, one should be able to model differences between two versions of a model. This can of course be done by using lexical differencing, as used for text files, on the data representation of the model. However, the result of such analysis is often not useful, as (1) the actual differences occur at the granularity level of nodes, links, labels and attributes and (2) models are usually not sequential in nature and equivalences between models converted to text will not be taken into account. Hence, model differencing should be done at an appropriate level of abstraction, and take semantics into account.

In addition to finding differences, one should be able to *represent* them explicitly as a model, called the *delta model*. There are two kinds of representations: operational and structural. In the operational (or change-based) representation [121], also used for detecting model inconsistencies [20, 19], the difference between two versions of a model is modelled as the series of CRUD (Create/Read/Update/Delete) edit operations that were performed on one model to arrive at the other [1, 86]. When these operations are recorded live from a tool, this strategy is very accurate and powerful, though dependent on that particular tool and difficult to manipulate explicitly. In the structural representation, the difference is encoded as a declarative model (*e.g.,* [30]). The two representations can typically be converted to one another, possibly with loss of information, such as the order in which changes were applied. In any case, these representations can be composed of small operations. These operations, which we will call *delta operations*, are reusable. In modelling language evolution, the metamodel is often the subject of change. Therefore, further in this chapter, we consider the delta model as the difference between two versions of a modelling language's metamodel.

When the metamodel of a modelling language evolves, the most prominent side-effect is that its instance models may no longer conform to the new metamodel. It is widely accepted that a model co-evolution (*i.e.,* migration) is best modelled as a model transformation [31, 77, 86, 212], which we will call the *migration transformation*. Since the migration transformation is directly influenced by the delta model, the migration transformation can be split up accordingly in *migration operations*, each linked to a delta operation [86].

Gruschko *et al.* [77] distinguish between non-breaking, resolvable and unresolvable delta operations:

— *non-breaking operations* do not break conformance of models with respect to the new metamodel, and hence do not require co-evolution. For example, the addition of a class in the metamodel is a non-breaking operation. Because of the optional nature of the change, none of the instance models have to be co-evolved in order to conform to the new metamodel. The instance models simply do not make use of the new language feature;

— *breaking and resolvable operations* cause inconsistencies that can be resolved by automated co-evolution. For example, renaming a class in the metamodel requires renaming of the class instances in the instance models;

— Model co-evolution for *unresolvable operations* requires additional information in order to execute. For example, when an attribute is added to a class in the metamodel, a new attribute is created for each instance of that class. However, the initial value of this feature is unknown, as it differs from instance to instance. There are two ways to solve this problem. On the one hand, a default value or expression can be given. Although this default value must be manually included in the migration transformation, the instance model can be migrated fully automatically. However, this does restrict the migration to one default scenario, which might not be accurate for each co-evolving artifact. On the other hand, one could manually adapt each model. Though the only correct solution in some cases, it can be a tedious job to perform manual co-evolutions on each model separately.

Cicchetti *et al.* [31] classify changes as additive, subtractive and updative, referring to CRUD operations. There is no consensus about the exact meaning of these terms in the literature. In some work, the terms refer to adding, removing or updating elements in the metamodel. We however define these terms in the context of the metamodel's semantics, *i.e.,* the effect of the change on the modelling space. After additive changes have been applied, more models are valid. In other words, the modelling space of the language is enlarged. Subtractive changes reduce the modelling space, and updative changes ensure the same size. With these operations, evolution of the part of the metamodel that specifies the static semantics can be expressed. If necessary, similar operations can be created for the constraint language, as it also has a metamodel [170].

From all introduced concepts, we create a classification of instance model migration.

— ***Ad hoc transformation vs. coupled transformation.*** As previously mentioned, migration transformations are naturally modelled as model transformations. A migration transformation can be composed in a non-structured, ad hoc manner, or by coupling it to the metamodel changes. For coupled transformation, different kinds of changes require different migration operations, which can be classified as follows:

— ***Non-breaking vs. breaking changes.*** Non-breaking and breaking changes can be distinguished. Breaking changes are often divided into the following two types:

— ***Resolvable vs. unresolvable changes.*** Resolvable changes can be automated. According to many authors, changes can be unresolvable, meaning that the according migration operations must be created manually [31, 77, 86, 209, 212].

— ***Additive, subtractive and updative changes.*** Changes are additive if they increase the modelling space, subtractive if they decrease the modelling space, and updative if the modelling space does not change in size.

Orthogonal to this classification, there is the matter of execution of the migration transformation.

— ***Automatic migration execution vs. manual intervention.*** According to some authors, not

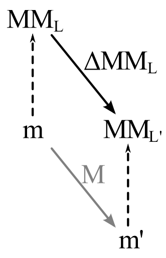


Figure 3.1: Models m have to be migrated when MM_L evolves.

only the creation of migration operations, but also the execution of the migration transformation can require manual intervention [31, 77]. Such a manual intervention can be different for each instance model.

All of these choices are incorporated into our approach.

3.2.2 De-constructing Evolution Actions

In this section, evolution as an operation is de-constructed into manageable, reusable parts based on the literature. Since the driving force is language evolution, we will, similar to the literature, address evolution actions of the language's structure, *i.e.,* the metamodel. Later in Section 3.2.4, we will zoom out and look into the consequences of language evolution in the DSML relational ecosystem.

Figure 3.1 illustrates the singled out language evolution problem at a glance. Note that we visualise evolution by adding "depth" to the diagram. Full arrows are transformations, dashed arrows indicate conformance (*i.e.,* that a model conforms to the language constraints). After evolution ΔMM_L of a metamodel MM_L, models m, which conform to MM_L, need to be migrated to m', which conform to $MM_{L'}$, by creating a suitable migration *M*.

As described in related work in Section 3.5, in the literature ΔMM_L is described as a sequence of delta operations. For models that are not metamodels, similar operations can be defined as the set of possible evolution actions on the abstract syntax of the model. As in the literature, the delta operations are classified as non-breaking, resolvable breaking and unresolvable. Table 3.1 shows how the impact of delta operations can also be classified as updative, additive or subtractive, referring to CRUD (create/read/update/delete) operations. For each of the delta operations, there is a default migration operation that can be used to co-evolve instance models. As expected, the non-breaking operations do not need a migration operation, the resolvable operations have a useful migration operation, and the unresolvable operations have a default operation that does not always have the desired migration effect. Additionally, each delta operation has an inverse operation which we will need in our approach. By definition, performing a delta operation on a metamodel, followed by its inverse operation, results in the original metamodel. Note that the same is not necessarily true for its migration operation. Furthermore, each delta operation is operationalised by a function, with parameters and preconditions that must be met to execute the delta operation. An example of a delta operation definition is expressed as follows:

Delta operation	Type	Migration Operation	Inverse Operation
Non-breaking delta operations			
Generalise property	Additive	None	Restrict property
Add class	Additive	None	Eliminate class
Add non-obligatory property	Additive	None	Eliminate property
Make class concrete	Additive	None	Make class abstract
Extract abstract superclass	Updative	None	Flatten abstract hierarchy
Extract superclass	Additive	None	Flatten hierarchy
Flatten abstract hierarchy	Updative	None	Extract abstract superclass
Push property from abstract class	Updative	None	Push property from abstract class
Pull property to abstract class	Updative	None	Pull property to abstract class
Breaking and resolvable delta operations			
Eliminate class	Subtractive	Eliminate instances	Add class and Add non-obligatory property
Eliminate property	Subtractive	Eliminate instances	Add (non-)obligatory property
Make class abstract	Subtractive	Eliminate instances	Make class concrete
Extract class	Updative	Extract properties and add instances	Inline class
Inline class	Updative	Inline properties and remove instances	Extract class
Flatten hierarchy	Subtractive	Eliminate superclass instances	Extract superclass
Push property	Subtractive	Eliminate properties from superclass instances	Pull property
Rename class	Updative	Change instances	Rename class
Rename property	Updative	Change instances	Rename property
Breaking and unresolvable delta operations			
Add obligatory property	Additive	Add default instances	Eliminate property
Pull property	Additive	Add default properties for superclass instances	Push property
Restrict property	Subtractive	Remove instance if non-compliant	Generalise property

Table 3.1: Delta operations adapted from [31] and [86], extended with their migration operations and inverse operations.

Add obligatory property

Description: Add a new property (attribute or association) with lower multiplicity of at least 1
Type: additive
Impact: breaking and unresolvable
Migration operation: add property instances with default values
Inverse operation: Eliminate property
Function signature: `AddObligatoryProperty(name, owner, lowsrc, highsrc, type[, lowdst, high-dst][, default])`
Parameters:

— `name` (string): name of the new property

— `owner` (class): the owner of the property

— `lowsrc` (integer): the lower bound at the source

— `highsrc` (integer): the upper bound at the source

— `type` (type or class): the target (in case of an association) or type (in case of an attribute) of the property

— `lowdst` (integer): (in case of an association) the lower bound at the target

— `highdst` (integer): (in case of an association) the upper bound at the target

— `default` (any): (in case of an attribute, optional) the default value

Preconditions:

— `highsrc` > 0, `highdst` > 0, `lowsrc` $\leq$ `highsrc`, `lowdst` $\leq$ `highdst`, `lowsrc` > 0 or `lowdst` > 0

— `default` is of type `type`, or none

— `name` is not yet taken by another of `owner`'s properties

A full list of definitions of all delta operations can be found in Appendix B.

The evolutions listed in Table 3.1 represent manipulations that typically occur on a given metamodel, more specific than basic CRUD operations. The table also lists complex evolutions like "flatten hierarchy" (eliminating a superclass and adding all its properties to the subclasses) or "generalise metaproperty" (relaxing the cardinality of a property). In those cases, the evolution could also be seen as the composition of simple changes, but it captures its full meaning when considered as a single adaptation step. Subsequently, a meaningful migration operation can be defined. If the metamodel contains static semantics, in the form of*e.g.,* OCL constraints [152], similar operations can be contrived. However, this is outside the scope of this thesis, and is left for future work.

The classification proposed above highlights the criticality of the metamodel evolution detection and representation in order to achieve a profitable migration of the existing instances. (Meta-)model comparison has been an active field of research; it is an intrinsically complex task since it has to deal with graph isomorphisms, *i.e.,* with the problem of finding correspondences between two given graphs. In this chapter we assume that the metamodel evolution, *i.e.,* ΔMM_L in Figure 3.1, is given, as reflecting the language engineer's intentions, in terms of the operations classified in Table 3.1: it could be obtained as directly traced from a tool, or encoded by hand. For our approach, both techniques are applicable.

3.2.3 The Projection Problem

When a language *L* evolves to *L'*, *M* is created to migrate instance models. Ideally, the image of the migration transformation *M* is equal to the evolved language *L'*. Unfortunately, it is not always the case that the migration transformation projects the original language to the evolved language. We call this phenomenon the *projection problem*, which is illustrated in Figure 3.2 (a). Ellipses are languages, or sets of models, conforming to the denoted metamodel. Arrows are transformations. A language *L* evolves to *L'* by an evolution ΔMM_E, and instance models such as *m* are migrated by *M* (which is related to ΔMM_E) to m_M. There are two different cases where $M(L) \neq L'$, namely $L' \setminus M(L) \neq \emptyset$ (*e.g.,* for *m'*) and $M(L) \setminus L' \neq \emptyset$ (*e.g.,* for *m''*). The case of $L' \setminus M(L)$

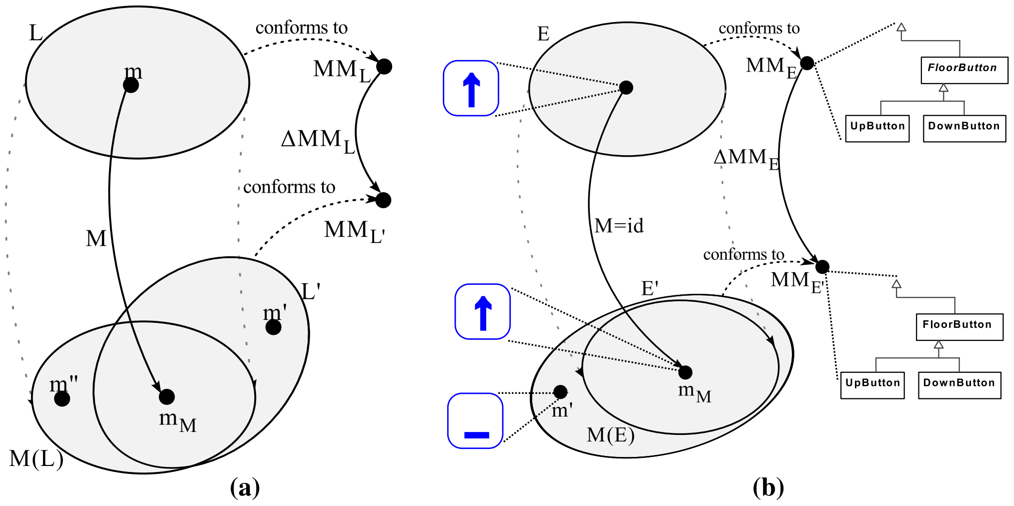


Figure 3.2: Set representation for evolution with (a) the projection problem after migration and (b) example of the case of $L' \setminus M(L)$.

occurs for additive changes. Models like *m'* are valid but cannot be the result of the execution of *M*. In other words, *M* does not exactly project *L* to *L'*. This can be illustrated by the fact that non-breaking, additive changes do not require migration at all. In that case $M(L) = L$ and $L \subset L'$, so $M(L) \subset L'$. The other case of $M(L) \setminus L'$ only occurs if *M* is not well implemented. Models like *m''* are models that are the result of the execution of *M*, but are not valid in the new language *L'*. For example, this would happen if no proper *M* is composed for a subtractive change. This is a matter of faulty implementation instead of an inevitable structural problem. Therefore, we will only take the case of $L' \setminus M(L)$ into account for the projection problem.

Figure 3.2 (b) shows an example of the projection problem resulting in models that are in $L' \setminus M(L)$. The same layout is used as in Figure 3.2 (a). Suppose that in the *Elevator* metamodel as shown in Figure 1.4 (visualised partially) the class FloorButton is made concrete as a non-breaking, additive change "make class concrete". The delta model ΔMM_E can thus be modelled as `MakeClassConcrete(FloorButton)`. Since the change is non-breaking, *M* is the identity transformation *id* that applies no changes at all to the model. The model in *L* is thereby migrated to the model in *M(L)*, and its structure is preserved. However, new models are possible in $L' \setminus M(L)$ that include instances of FloorButton (represented as buttons with a floor icon inside). The fact that this model is out of the scope of *M*, will add some restrictions on the use of *M*.

3.2.4 De-constructing Evolution Consequences

As discussed in Section 3.1, there are many possible situations in which co-evolution can occur. These possibilities can be mapped onto the DSM relational ecosystem of Figure 1.1, and in the *Elevator* example onto the *Elevator* relational ecosystem of Figure 1.3. When we look more closely at the possible scenarios, similarities can be distinguished in the steps that are taken in the co-evolution scenarios. As consistency and continuity are desired during system evolution, co-evolution is in fact the (chained) relationship between metamodel, model and transformation

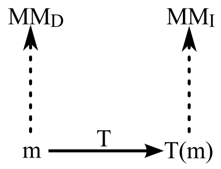

$$\begin{tikzcd}
MM_D & MM_I \\
m \arrow[u, dotted] \arrow[r, "T"] & T(m) \arrow[u, dotted]
\end{tikzcd}$$

Figure 3.3: Basic co-evolution schema.

(instead of only metamodel and model, as suggested in related work). Thereby, it is important to note that unidirectional transformations have a domain (*i.e.,* the language that is translated) and an image (*i.e.,* the language this domain is translated to). In other words, for language evolution, both *incoming* and *outgoing* transformations must be dealt with separately. This distinction does not have to be made in case bidirectional transformations are used, but in practice, unidirectional transformations are often preferred as bidirectional transformations come with technological difficulties. As a consequence, we can reduce the problem of co-evolution to the diagram in Figure 3.3, where a domain metamodel MM_D and an image metamodel MM_I, and their instance models are distinguished, with a transformation T in between. Again, arrows denote transformations and dotted arrows denote "conforms to"-relationships. The diagram reduces the problem of evolution to its bare essence.

The possible evolution scenarios can be broken down into a few basic ones which are depicted in Figure 3.4. Small, dashed arrows denote a (semi-)automatic generation. Each diagram starts from the relation between domain and image metamodels, given in Figure 3.3. With these four basic scenarios, every possible evolution of Figure 3.3 is covered. These are explained in detail in the following paragraphs.

3.2.5 Model Evolution

Figure 3.4 (a) shows model evolution. Although in the diagram m evolves, model evolution occurs in fact when either m or $T(m)$ evolves. This kind of evolution is not a language evolution and is included for completeness, and has quite simple consequences for other artifacts in the relational ecosystem. Some model m evolves to m' (still conformant to MM_D). In step 1 (the only step), a delta model Δm is constructed (either automatically or manually) that models the evolution of m to m': $m' = m + \Delta m$. The evolution itself is typically represented as a migration transformation M, which is the identity transformation for all models in language D, with the exception of m. If the model is the input of a transformation, as depicted in Figure 3.4 (a), it is desirable that this transformation is executed again. The transformations whose execution results in the model (in Figure 3.4 (a) this would mean that $T(m)$ evolves rather than m) do not have to be executed again.

In the context of the Elevator relational ecosystem, Elevator models could evolve via the insertion, removal or redirection of Floors, Buttons and Elevators. Changing m (as shown in Figure 1.1) this way might be desirable that some of the transformations are re-executed. For example, if one wants to use a Petri net model for analysis, the semantic mapping $[[.]]_{TS}$ must be applied again. In general, κ_i and $[[.]]_{TS}$ and/or $[[.]]_{OS}$ transformations, as well as any other transformation T_j from model m, remain valid since no metamodel evolution has occurred, but have to be re-executed. No further co-evolution is required.

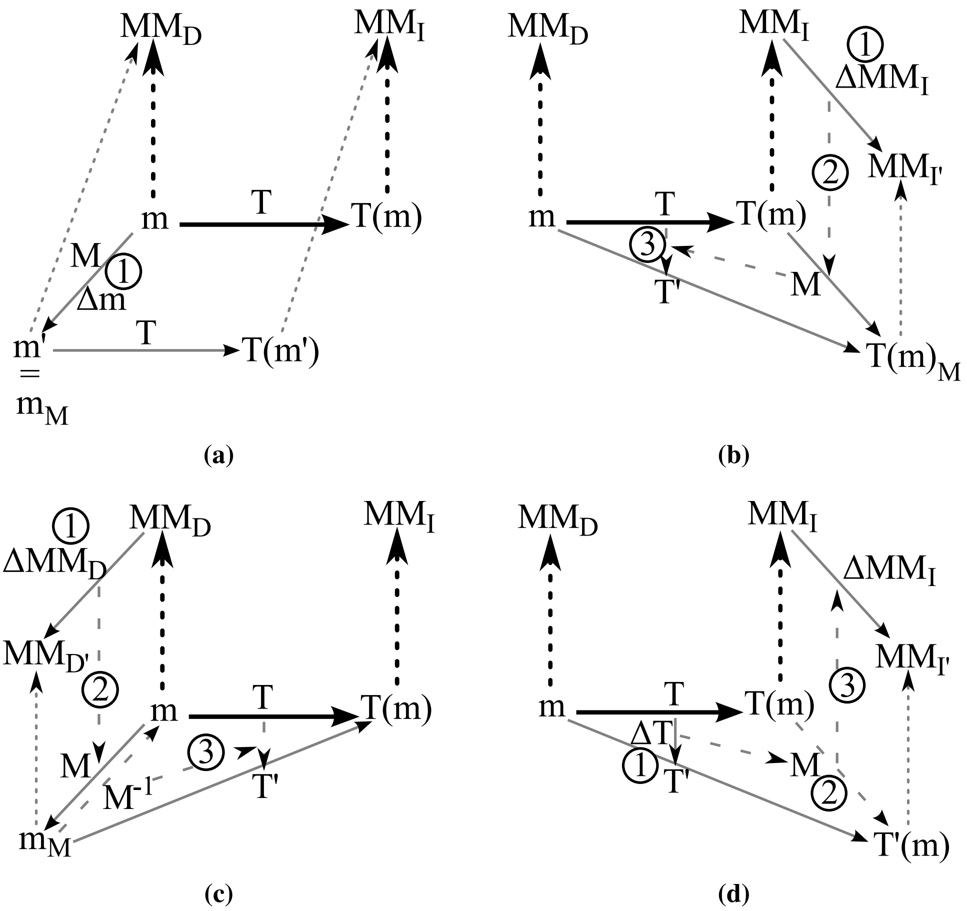


Figure 3.4: Co-evolution in (a) model evolution, (b) image evolution, (c) domain evolution and (d) transformation evolution.

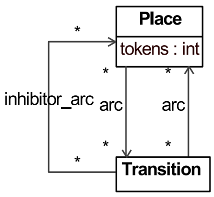


Figure 3.5: A metamodel for the Petri net formalism, including inhibitor arcs.

3.2.6 Image Evolution

Image evolution is shown in Figure 3.4 (b). Suppose that a metamodel MM_I evolves to $MM_{I'}$. In the context of the *Elevator* relational ecosystem of Figure 1.3, a possible image evolution is the evolution of the target metamodel for the semantic mapping transformation (in this case, Petri nets). We illustrate this with an evolution of the Petri net metamodel of Figure 1.9. Suppose

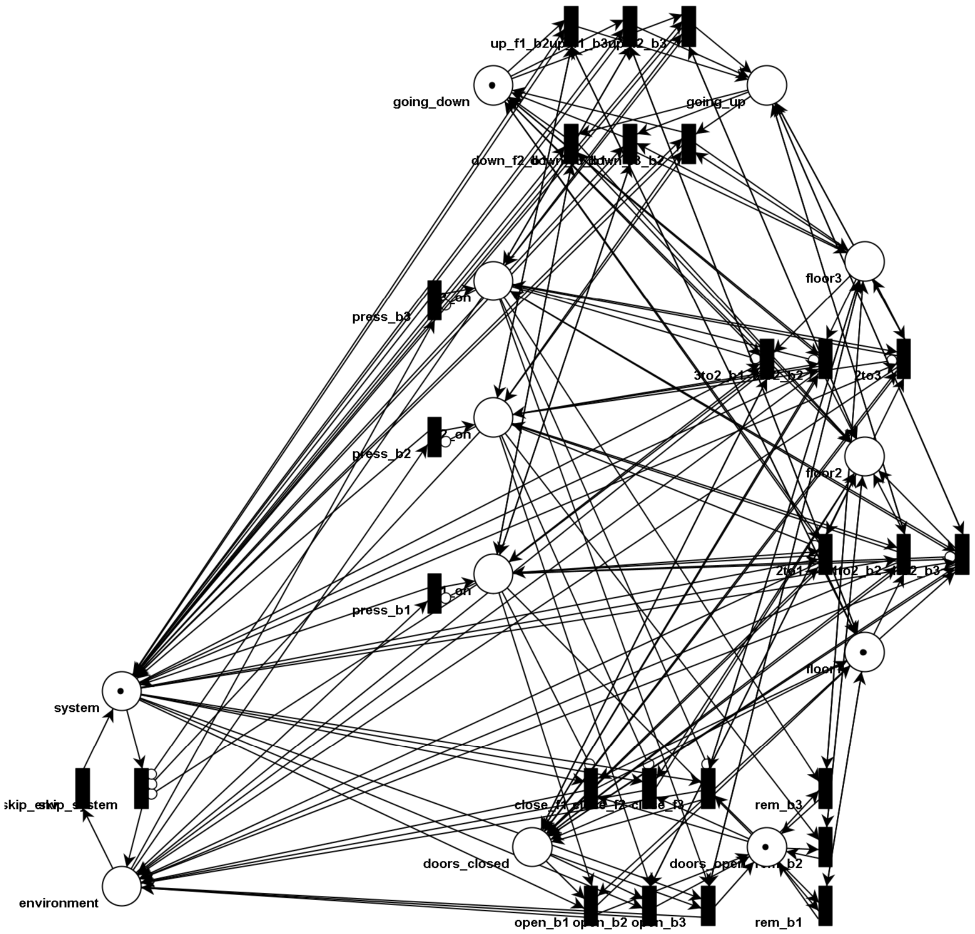


Figure 3.6: The *Elevator* system in inhibitor Petri nets.

that we add the concept of inhibitor arcs to the Petri net formalism, as shown in Figure 3.5. This is done by the non-breaking, additive "add non-obligatory property" delta operation. Conceptually, inhibitor arcs represent constraints on a transition, saying that the connected place should not contain a token. The addition of inhibitor arcs makes the Petri net formalism Turing complete [216]. In step 1 of the evolution, a delta model ΔMM_I is constructed to represent the difference between metamodels MM_I and $MM_{I'}$. In the example, ΔMM_I can be represented as the delta operation `AddNonObligatoryProperty(name="inhibitor_arc", owner=Place, highsrc=*, type=Transition, highdst=*)`. In concrete visual syntax, the inhibitor arc is usually represented as an arc with a circle at its end. However, we will focus here on the evolution of the metamodel only.

In step 2, a migration transformation M is generated from ΔMM_I. The execution of M co-evolves models $T(m)$ to models $T(m)_M$ such that they conform to the new metamodel $MM_{I'}$. In the simple example, this can be skipped because the change is non-breaking. We added inhibitor arcs for a reason however, and by using inhibitor arcs we can define M as the replacement of

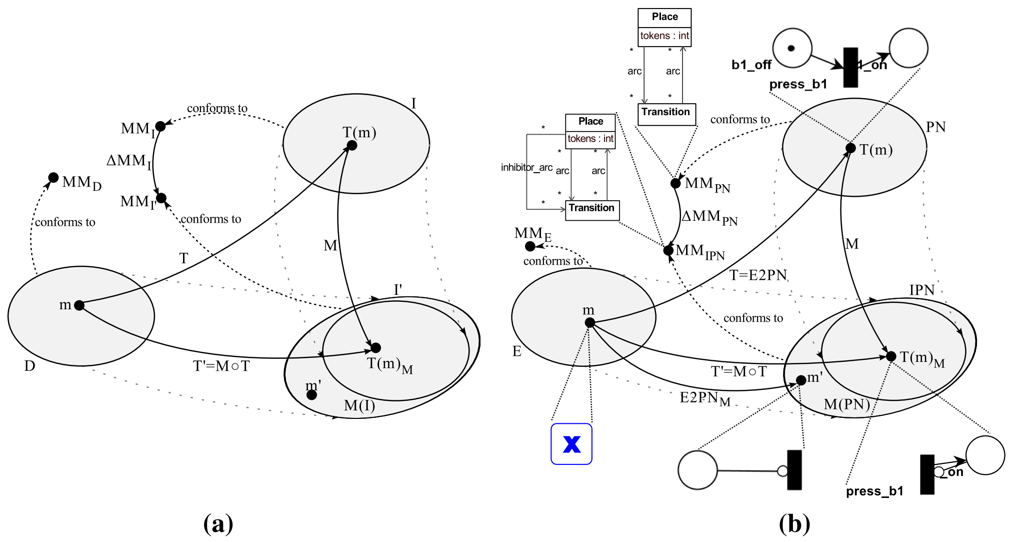


Figure 3.7: The projection problem in image evolution: (a) $T' = M \circ T$ does not hold and (b) an example.

place-antiplace patterns with only one Place. Every outgoing arc of the antiplace is replaced with an inhibitor arc from the related place. Applying the migration transformation on $[[m]]_{TS}$ representing the three-floor *Elevator* system shown in Figure 1.11 results in the inhibitor Petri net shown in Figure 3.6. The representation of Buttons has changed.

Moreover, the execution of transformation T has to result in valid models (*i.e.,* models that conform to $MM_{I'}$). Consequently, T is migrated to a new transformation T' (step 3), which is able to transform every possible model m conforming to MM_D to an appropriate model $T(m)_M$ conforming to $MM_{I'}$. The diagram presents an intuitive solution for the generation of this T': for every model m, $T'(m) = M(T(m))$ holds, and thus we have $T' = M \circ T$. Hence, the co-evolved transformation T' can be obtained simply by composing transformations T and M. The trivial way of doing this is to execute T, followed by executing M on the output model of T. Optimising this composition is a challenge in its own right. In our evolution example, the *E2PN* transformation can be migrated without digging into the transformation model: each *Elevator* model can be transformed to the new version of Petri nets by first applying *E2PN*, resulting in an old version Petri net model, followed by the execution of the migration transformation M, which transforms models of the old version to the new version. Resulting models will conform to the evolved Petri net metamodel. Of course, this setting is only valid for simple language evolutions. $T' = M \circ T$ does not hold for the entire evolved image I'.

Because of the projection problem, $T' = M \circ T$ does not always hold for image evolution. Following Figure 3.3, Figure 3.7 (a) shows a model m conforming to MM_D that can be transformed by T resulting in the model $T(m)$ conforming to MM_I. The image MM_I evolves to $MM_{I'}$ by an additive operation, which results in I' being larger than $M(I)$. If T' is acquired by executing $M \circ T$, it turns out that semantic information is not taken into account, as the model is transformed to the language I first, which does not include the newly added semantic information. As explained in Section 3.2.3, it is generally desirable that the evolved transformation T' can explore the full power

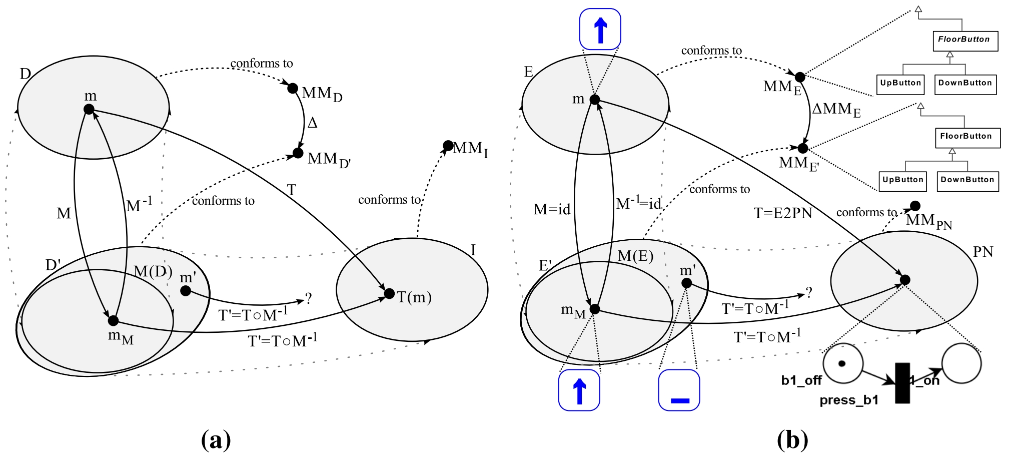


Figure 3.8: The projection problem in domain evolution: (a) $T' = T \circ M^{-1}$ does not hold and (b) an example.

of the new version of the image language *I'*.

In Figure 3.7 (b) an example is shown. The example is based on the evolution from Petri nets inhibitor Petri nets of Figure 3.5. Now, the *E2PN* transformation is included with *Elevator* as domain. If it is stated that $T' = M \circ T$, then *T'* transforms the *Elevator* model to the model in *M(PN)*, by first executing the original *T*, followed by migrating the model in *PN*. However, this way, some changes that were deliberately applied to the Petri net language are ignored. In this case, only place-antiplace patterns will be covered by *M*. Inhibitor arcs in general are not supported by $T' = M \circ T$. Despite the projection problem, the principle of $T' = M \circ T$ will prove to be useful in our final approach, as presented in Section 3.3.

3.2.7 Domain Evolution

Figure 3.4 (c) shows domain evolution, where MM_D evolves. The artifacts that co-evolve are similar to those in the image evolution scenario. However, co-evolved transformations *T'* can now be obtained by $T' = T \circ M^{-1}$ which implies the need to construct an inverse transformation M^{-1}. More specifically, in order to transform models that conform to the new version of the language to the image language, they are first co-evolved *back* to the old version, and then the original transformation can be applied. Again, the transformation does not have to be adapted. In the context of the *Elevator* relational ecosystem, domain evolution occurs when the *Elevator* metamodel of Figure 1.4 (a) changes.

In domain evolution, the projection problem can also occur, as shown in Figure 3.8 (a). Again, during transformation where $M(D) \neq D'$, critical information can be lost when migrating the model back to its old version using M^{-1}. Again, this occurs for additive changes. Also, it is desirable that the transformation *T'* can be applied to its entire domain, including $D' \setminus M(D)$. This part of the transformation cannot be obtained by $T' = T \circ M^{-1}$ if $M(D) \neq D'$. One might intuitively think that subtractive changes also result in a projection problem, as for these changes new information should be "invented" when executing M^{-1}. However, M^{-1} typically does not have to extend the models again to their original form because the language was reduced deliberately. An example of the projection problem for domain evolution is shown in Figure 3.8 (b), which continues from the example of Figure 3.2 (b). Since the change is non-breaking, *M* and consequently M^{-1} are the identity transformation *id*. Similar to image evolution, $T' = T \circ M^{-1}$ applies for models in *M(E)*. However, models in $E' \setminus M(E)$ that use *FloorButton* instances (which are visualised by an empty button icon) cannot be transformed by *T'*. This is problematic, as it should be possible that every possible *Elevator* model is transformed to a Petri net.

3.2.8 Transformation Evolution

Figure 3.4 (d) shows transformation evolution. For example, in the context of the *Elevator* relational ecosystem, the semantic mapping *E2PN* could be adapted in such a way that the number of button presses is recorded in a place. If transformations evolve according to a delta model ΔT, it is possible that they only have to be executed once again. In this case, the changes on the transformation are limited: the image of the evolved transformation *T'* must conform to MM_I (*i.e.,* the existing Petri net language). As previously discussed, other artifacts such as the image metamodel might also co-evolve. In such cases, a migration transformation *M* must be constructed from which a delta model ΔMM_I can be derived. In the *Elevator* example, one could add a time delay to opening and closing the doors, or changing floors. In this case, a timed Petri net could be needed as a result of a change in *E2PN*. Note that such a decision on the co-evolution of the metamodel highly depends on the intention of the evolution and is therefore done manually. The construction of the migration transformation *M* helps in this process, but can be left implicit at this stage. In case of co-evolution of the domain however, if left implicit, *M* will be obtained in a later stage when the scenario of metamodel evolution is carried out. Consequently, the issue of the combination of evolution scenarios arises, which is discussed in general in the following section.

3.2.9 Evolution Scenario Amalgamation

Using a combination of these basic four scenarios of Figure 3.4, all possible evolutions can be carried out. For unresolvable changes, a general *M* cannot be found. Note also that the projection problem exists, so automated co-evolution is not always possible. Due to the projection problem, transformations have to support the models in $L' \setminus M(L)$.

In the *Elevator* relational ecosystem, it is conceivable that a language engineer desires domain and image evolutions. Consider the example evolutions of domain evolution (making FloorButton concrete), in combination with image evolution (adding inhibitor arcs). The co-evolution of the instances of both formalisms is independent of each other. The *E2PN* transformation, however, co-evolves for both evolutions. After both are applied, the co-evolved transformation will again be obtained by following the evolution diagram as follows: $T'(m) = M_{\Delta MM_{PN}}(T(M^{-1}_{\Delta MM_E}(m)))$. In general, evolutions can be chained in transformation co-evolution. Of course, the projection problem still applies.

Simply combining basic evolution scenario solutions to solve complex scenarios may yield sub-optimal results. Firstly, domain and image evolutions might happen at the same time because they are closely related. In the example of domain and image evolution, it can be desirable that the *E2PN* transformation is only migrated once, taking both evolutions into account at the same time. If not, some artificial mappings might be necessary when the *E2PN* transformation is migrated for only one of the evolutions. Secondly, the performance of combining scenarios might be sub-optimal. In a system under development, an additional transformation must be executed for each additional evolution. Hence, the total number of migration transformations to be executed after an evolution is $\mathcal{O}(n{\cdot}v)$, with n the number of artifacts that are co-evolved and v the number of existing versions. It may be possible to obtain the same output models while executing less and/or simpler transformation rules. An open research problem is the (semi-)automatic merging of sequences of transformations into more efficient (but equivalent) transformations. Hereby, issues such as order dependence of transformation rules must be overcome. Because they are more modular, at a suitable level of abstraction (hiding matching and rewriting issues), and explicitly metamodelled (allowing for higher-order transformations) it is likely that rule-based transformation specifications will be more amenable to such optimisations than pure code-based ones. Nevertheless, complex rule pattern dependence analysis [84, 132] has to be conducted in order to reason about the composability of transformation rules.

3.3 Re-constructing Evolution

In Section 3.2, evolution was de-constructed into manageable basic scenarios. The delta transformation was divided into atomic steps and the relational ecosystem of Figure 1.1 was broken down into the basic schema of Figure 3.3. In this section, we use this breakdown to re-construct all possible evolution scenarios. Then, similarities and differences between scenarios can be investigated and a suitable solution in the form of instance and transformation co-evolution can be customised to maximise automation.

In the basic co-evolution schema of Figure 3.3, each of the involved artifacts can evolve, causing some other artifacts to co-evolve. Hence, we assume that each relational ecosystem can be mapped onto (multiple instances of) this schema, which originated from the modelling artifacts shown in Figure 1.1. An instance of the basic schema can be identified in a relational ecosystem by identifying a transformation as an instance of T. Then, incoming models are instances of m and their metamodel is an instance of MM_D. The resulting models are instances of $T(m)$ and their metamodel is an instance of MM_I. However, in theory, a transformation has one input model and one output model, this input or output model can be a modular composition of two models (possibly conforming to two different metamodels) so that in practice transformations can have multiple input and/or output models. An example of a transformation with a composed input model is model merging. An example of a transformation with a composed output model is executing the operational semantics in *ProMoBox* which generates a model in the output language and a model in the input language. Note that *ProMoBox* operational semantics also has a composed input model, conform to the run-time and input language. In many existing transformation tools, the transformations have combinations of more than one metamodel describing input or output. In those cases, it is desirable to maintain the modularity of the different formalisms that are used. Therefore, each input or output metamodel is mapped onto a different instance of the basic schema. For example, a model merging transformation has two mappings to the basic schema: one with the metamodel of the first input model as MM_D, and one with the metamodel of the second input model as MM_D.

Additionally, we must take into account the type of transformation T. Different kinds of transformations have different co-evolution consequences. As explained in Section 1.2, transformations

	Changing artifact				
Type of T	MM_D	MM_I	T	m_D	m_I
unidirectional exogenous transformation	co-evolve m_D and T, execute T	co-evolve m_I and T	change MM_D and/or MM_I	change MM_D	change MM_I
bidirectional exogenous transformation	co-evolve m_D and T and T^{-1}, execute T	co-evolve m_I and T and T^{-1}, execute T^{-1}	change MM_D and/or MM_I	change MM_D	change MM_I
unidirectional endogenous transformation	co-evolve m_D/m_I and T, execute T	co-evolve m_D/m_I and T	change MM_D/MM_I	change MM_D/MM_I	change MM_D/MM_I
bidirectional endogenous transformation	co-evolve m_D/m_I and T and T^{-1}, execute T	co-evolve m_D/m_I and T and T^{-1}, execute T^{-1}	change MM_D/MM_I	change MM_D/MM_I	change MM_D/MM_I
translational semantics $[[\cdot]]_{TS}$	co-evolve m_D and $[[\cdot]]_{TS}$, execute $[[\cdot]]_{TS}$	co-evolve m_I and $[[\cdot]]_{TS}$	change MM_D and/or MM_I	change MM_D	change MM_I
operational semantics $[[\cdot]]_{OS}$	co-evolve m_D/m_I and $[[\cdot]]_{OS}$, execute $[[\cdot]]_{OS}$	co-evolve m_D/m_I and $[[\cdot]]_{OS}$	change MM_D/MM_I	change MM_D/MM_I	change MM_D/MM_I
concrete mapping function κ and π (rendering and parsing)	co-evolve MM_I, m_D and κ and π, execute κ or π depending on changing artifact	co-evolve MM_D, m_I and κ and π, execute κ or π depending on changing artifact	change MM_D (and consequently MM_I)	change MM_D (and consequently MM_I)	change MM_D and (and consequently MM_I)

Table 3.2: Evolution scenarios for the basic co-evolution schema.

can be (orthogonally) endogenous/exogenous, unidirectionally/bidirectionally, in-place/out-place [133]. Additionally, transformations can be used as parsing or rendering functions, semantic mapping or model-to-model transformation [110]. They can also keep generic links between concepts in different models, which results in traceability between concepts of the model. Although this can greatly facilitate the automation of co-evolution at the implementation side, this is irrelevant from the architectural point of view presented in this section. Also note that transformation models can range from simple search-replace scripts, to XSLT transformations, to rule-based graph transformations. Although the above distinction suffices to express the different kinds of transformations, the concrete mapping functions κ and π can be considered a special case because of the tight coupling between the metamodels of abstract and concrete syntax. This relation is homomorphic in nature. In order to maintain that homomorphic relation, a metamodel evolution (*e.g.,* of the abstract syntax) that changes the metamodel structurally requires migration of the other metamodel of the relation (*e.g.,* of the concrete syntax). In Table 3.2, the possible co-evolution scenarios are explored, where rows represent the kind of transformation, columns represent the artifact that changes. As artifacts of Figure 3.3 are used, we assume that every language evolution in a system can be mapped onto one or more cells in this table.

Some conclusions can be drawn from observing the table. The default scenario is the evolution of MM_D or MM_I, *i.e.,* the first two columns. Evolutions of T, m_D or m_I usually do not cause a change on the metalevel (*i.e.,* the level of the language). Indeed, evolution causes at most execution of T, in order to keep the models in the system consistent. This is in particular true, and usually must happen instantly, for multi-view systems [78]. As this does not change anything at the metalevel, it is not taken into account. In some cases, however, a change at the metalevel is desirable. Suppose that a domain user can adapt a language by changing the model. This is useful because the domain user can perform simple modifications to a language without extensive knowledge of language engineering, such as metamodelling or transformation modelling (in case of adaptation of T itself, the user must understand it of course). Instance-based modification of a language is possible if the modelling environment allows *bottom-up* editing, where metamodels are extracted from instances [141, 40]. The last three columns in Table 3.2 represent such scenarios where the language is adapted, as we are interested in language changes.

A second conclusion is that, apart from the concrete mapping, all types of T are very similar. Evolution of a language always leads to co-evolution of instances and transformation models, with a possible execution of T as a consequence. Note that we limit each scenario to the schema of Figure 3.3, so, some real-life scenarios might trigger more than one evolution scenario of Table 3.2. For example, when evolving a metamodel, each of the related transformation models is co-evolved, which can each be mapped onto a table cell. Furthermore, because of the co-evolution consequences, a chain of transformation executions can be triggered.

The concrete mapping is, however, notably different. The abstract syntax must be kept consistent with the concrete syntax(es). The concrete mapping functions κ and π (for rendering and parsing) can for this purpose be considered a special kind of transformation. Some tools such as *AToMPM* or *Fujaba* [72] force the user to explicitly implement a concrete mapping. Abstract syntax concepts can be instantiated using an Object Diagrams-like language, like in *metaDepth*. Tools such as *AToM*^{3} [46] include concrete syntax in the language definition (*i.e.,* attached to metamodel elements). This illustrates the tight coupling as well as the simple one-to-one nature of the concrete syntax mapping.

The remainder of this section will dig deeper in the an actual co-evolution step and will discuss the re-construction of metamodel evolution and instance migration in an isolated context. Whenever a change Δ is operated on a metamodel, a corresponding migration M should be operated on the existing instances. The creation of migration transformation is closely related to the changes on the metamodel however. Therefore, this section continues with an elaboration on the difference model composed from the delta operations presented in Section 3.2.2, which is a structured representation of the changes. Next, the creation of the migration transformation is presented for instance models, and it is explained how the presented technique is applicable to transformation models by incorporating the de-construction rules. The approach is demonstrated with a prototype in *metaDepth* in Appendix C.

3.3.1 Difference Model

Metamodel evolution can be modelled in a delta model as a sequence of delta operations, with associated migration operations, as presented in Section 3.2.2. Some migration operations are empty (in case of non-breaking delta operations), others can be implemented as simple scripts (*e.g.,* the migration operation of the breaking and resolvable delta operation “rename class”), but many more migration operations can be implemented intuitively as rule-based graph transformations. For unresolvable delta operations it is often not possible to define an automatic migration transformation, as the migration is instance-dependent. The goal is to alleviate both the language engineer and the domain user from as much manual work as possible when migrating instances by generating a modular migration transformation. The migration transformation can be adapted where necessary by the language engineer. Moreover, it can include manual intervention at execution time that needs to be performed by the domain user.

Reconsider the metamodel for the *Elevator* example of Figure 1.4. During the development cycle, a number of changes are made at the language level as shown in Table 3.3, consisting of delta operations from Table 3.1. The step-wise evolution is shown in Figure 3.9. The first delta operation limits the number of Buttons that requests a Floor to three using the "restrict property" operation. Then, an EmergencyButton is added by extracting a Button class in changes (operation 2 to 5). This is a button that, when pressed in case of emergency or by a technician, stops the elevator, and restarts the elevator when pressed again. Subsequently the UpButton and DownButton are replaced by a up_direction attribute in the now concrete class FloorButton in changes (operation 6 to 9). This way, FloorButton can be used for both directions (if necessary, their direction can even be changed), and is implemented as a single kind of button. Finally, the nr attribute is deemed redundant and is removed.

Concerning the stepwise evolution of the metamodel, we identify a problem related to the conformance of migrating instances. In most transformation tools, transformations such as the migration transformation require a well-defined input and output metamodel. As we want to keep the migration transformation as separate migration operations for modularity, we will need intermediate metamodels for each migration transformation. Indeed, the result of the first of many migration operations is not conform to the original version of the metamodel, and not yet to the metamodel of the new version of the language, but is conform to a metamodel "in between" the two versions.

To solve this problem, we suggest to use a relaxed *intermediate metamodel*, which is the result of merging the metamodels of two language versions. For a metamodel MM_{v0} and its evolution MM_{v1}, we define the intermediate metamodel as $MM_{[v0,v1]} = merge(MM_{v0}, MM_{v1})$. This intermediate metamodel is generated from the old metamodel, and the delta operations. In case of additive operations, the new elements are added to the intermediate metamodel as non-obligatory language concepts. In case of subtractive operations, the elements that need to be removed are instead made non-obligatory. In case of updative operations, both versions of the element are non-obligatory in the intermediate metamodel. The merging algorithm can be used recursively. For example if the evolution consists of three changes, then $MM_{[v0,v1]} = merge(merge(merge(MM_{v0}, \delta_1), \delta_2), \delta_3)$. Rename operations pose a slight problem: as it is inconvenient to give the element both names (the old name and the new name) in the intermediate metamodel. Therefore, only the updated language concepts are included. As a result, rename operations are shifted forwards in the delta model so that they are applied first, and their changes are applied if necessary to delta operation that were previously before the rename operation. As rename operations are not subject to the projection problem, changing their place in the delta model does not pose a problem.

The resulting delta model is shown in Table 3.3, where the "rename class" delta operation is shifted forwards and is now called δ_r. In this case, the rename did not affect other delta operations.

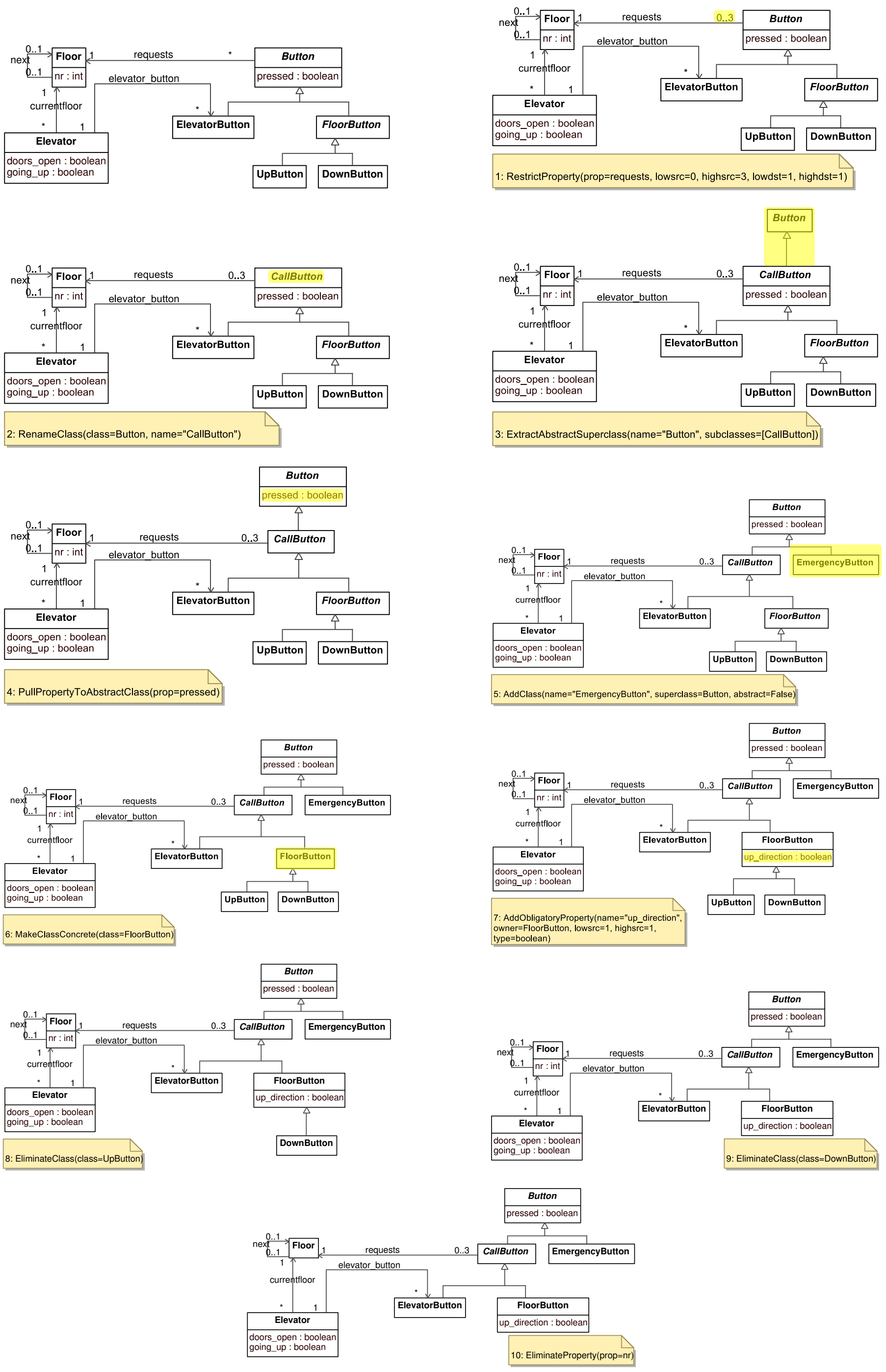

Floor
nr : int
next
0..1
requests
elevator_button
currentfloor
Elevator
doors_open : boolean
going_up : boolean
Button
pressed : boolean
ElevatorButton
FloorButton
UpButton
DownButton
1: RestrictProperty(prop=requests, lowsrc=0, highsrc=3, lowdst=1, highdst=1)
CallButton
2: RenameClass(class=Button, name="CallButton")
3: ExtractAbstractSuperclass(name="Button", subclasses=[CallButton])
4: PullPropertyToAbstractClass(prop=pressed)
EmergencyButton
5: AddClass(name="EmergencyButton", superclass=Button, abstract=False)
6: MakeClassConcrete(class=FloorButton)
up_direction : boolean
7: AddObligatoryProperty(name="up_direction", owner=FloorButton, lowsrc=1, highsrc=1, type=boolean)
8: EliminateClass(class=UpButton)
9: EliminateClass(class=DownButton)
10: EliminateProperty(prop=nr)

Figure 3.9: Evolution of the *Elevator* metamodel.

nr.	Operation	Type	Impact
δ_r	RenameClass(class=Button, name="CallButton")	Updative	Breaking and resolvable
δ_1	RestrictProperty(prop=requests, lowsrc=0, highsrc=3, lowdst=1, highdst=1)	Subtractive	Breaking and unresolvable
δ_2	ExtractAbstractSuperclass(name="Button", subclasses=[CallButton])	Updative	Non-breaking
δ_3	PullPropertyToAbstractClass(prop=pressed)	Updative	Non-breaking
δ_4	AddClass(name="EmergencyButton", superclass=Button, abstract=False)	Additive	Non-breaking
δ_5	MakeClassConcrete(class=FloorButton)	Additive	Non-breaking
δ_6	AddObligatoryProperty(name="up_direction", owner=FloorButton, lowsrc=1, highsrc=1, type=boolean)	Additive	Breaking and unresolvable
δ_7	EliminateClass(class=UpButton)	Subtractive	Breaking and resolvable
δ_8	EliminateClass(class=DownButton)	Subtractive	Breaking and resolvable
δ_9	EliminateProperty(prop=nr)	Subtractive	Breaking and resolvable

Table 3.3: The operational difference model Δ of the *Elevator* evolution.

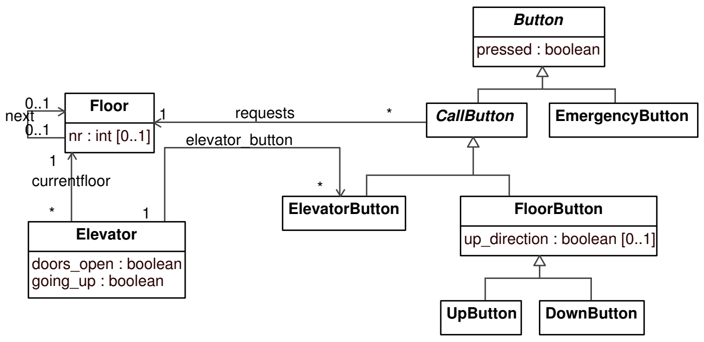


Figure 3.10: The intermediate metamodel of the *Elevator* metamodel and delta model of Table 3.3.

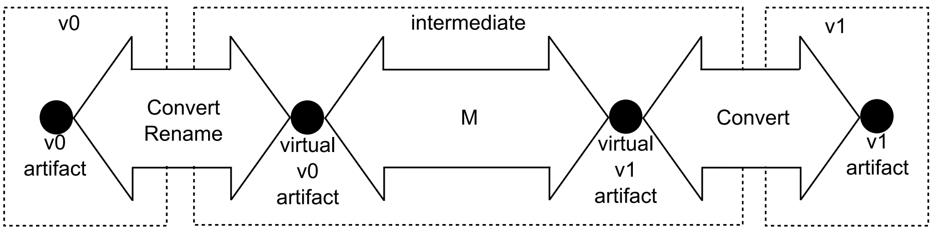


Figure 3.11: The transformation pipeline.

The intermediate metamodel taking changes δ_1 to δ_9 into account is shown in Figure 3.10.

As a result, every model that conforms to the old metamodel, as well as every model that conforms to the new metamodel, conforms to the intermediate metamodel. A step-wise co-evolution from an old version to a new version is now possible. Before and after execution of each migration step, the model under migration conforms to the intermediate metamodel.

In Figure 3.11, the step-wise migration of a version 0 artifact to a version 1 artifact is shown.

First, the version 0 artifact is converted to conform to the intermediate metamodel. This consists of replacing concepts of version 0 with their respective concepts of the intermediate metamodel. This is a purely technical step, and this often involves changing the package or path of every element's type. Often, this can be most efficiently done by using a script[1]. Additionally, type rename migrations are performed. The resulting artifact conforms to the intermediate metamodel, but is a "virtual version 0 artifact", as the artifact is semantically identical to its version 0 counterpart. In a sense, the virtual version 0 artifact lies at the edge of the modelling space described by the intermediate metamodel at the side of the version 0 metamodel. Next, the actual migration M is performed, by applying additive, subtractive and updative operations. After all these have been applied, the resulting artifact will be a "virtual version 1 artifact". Then, a straightforward conversion to a version 1 artifact is possible. As the bi-directional arrows suggest, it is possible to apply the migration in both directions. This way, the inverse migration transformation that can be used for domain evolution is also captured by Figure 3.11.

3.3.2 Migration of Instance Models

In this section an approach is presented to guide the user in solving co-evolution issues in a structured, step-wise manner in combination with systematically modelling the evolution (ensuring that the migrated models conform to the new metamodel). The approach distinguishes between syntactic (preserving consistency) and semantic (preserving continuity) migration. In-place transformation languages are employed. A migration is either automatically generated or adapted by the language engineer, and upon execution, manual intervention of the domain user might be required. The benefits of this technique are manifold, notably:

— the simplification of the migration specification by reusing the well-known graph transformation formalism of in-place transformations;

— the ability to express every possible evolution and migration by allowing graph transformation techniques;

— the reduction of the effort for the user by reusing generically applicable migration rules;

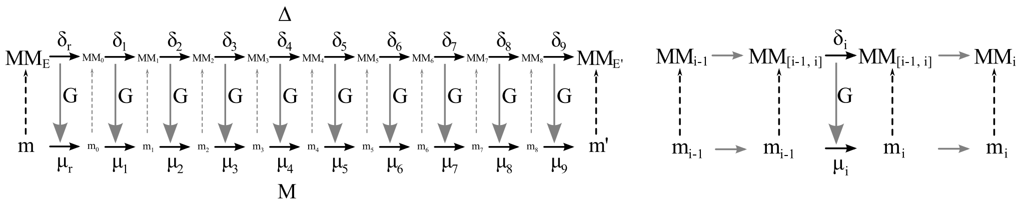
— the isolation of domain user intervention by singling out the delta operation;

— the control of domain user intervention by automated preventive and corrective mechanisms to validate that models conform to the language in each migration step;

— the optimisation of the migration execution itself by allowing in-place adaptation of the existing instances.

In short, we aim at a high degree of automation, a high degree of control, and high execution performance. Automation will reduce the effort for the language engineer and domain user, thus increasing productivity. Control will increase correctness of the migration process as well as facilitate the migration process for the language engineer at design time and for the domain user at execution time. Performance will affect scalability, and minimises the time to migrate a number of instance models. The migration process consists of three phases: automated synthesis, manual adaptation and execution.

[1]Note that in *ProMoBox*, similar conversion steps such as generating a run-time model from a design model are solved using such a script.



$$\begin{tikzcd}[column sep=tiny]
 & & & & & & & & & \Delta & & & & & & & & & & & \\
MM_E \arrow[rr, "\delta_r"] & {} \arrow[d, "G"] & MM_0 \arrow[rr, "\delta_1"] & {} \arrow[d, "G"] & MM_1 \arrow[rr, "\delta_2"] & {} \arrow[d, "G"] & MM_2 \arrow[rr, "\delta_3"] & {} \arrow[d, "G"] & MM_3 \arrow[rr, "\delta_4"] & {} \arrow[d, "G"] & MM_4 \arrow[rr, "\delta_5"] & {} \arrow[d, "G"] & MM_5 \arrow[rr, "\delta_6"] & {} \arrow[d, "G"] & MM_6 \arrow[rr, "\delta_7"] & {} \arrow[d, "G"] & MM_7 \arrow[rr, "\delta_8"] & {} \arrow[d, "G"] & MM_8 \arrow[rr, "\delta_9"] & {} \arrow[d, "G"] & MM_{E'} \\
m \arrow[u, dashed] \arrow[rr, "\mu_r"'] & {} & m_0 \arrow[u, dashed] \arrow[rr, "\mu_1"'] & {} & m_1 \arrow[u, dashed] \arrow[rr, "\mu_2"'] & {} & m_2 \arrow[u, dashed] \arrow[rr, "\mu_3"'] & {} & m_3 \arrow[u, dashed] \arrow[rr, "\mu_4"'] & {} & m_4 \arrow[u, dashed] \arrow[rr, "\mu_5"'] & {} & m_5 \arrow[u, dashed] \arrow[rr, "\mu_6"'] & {} & m_6 \arrow[u, dashed] \arrow[rr, "\mu_7"'] & {} & m_7 \arrow[u, dashed] \arrow[rr, "\mu_8"'] & {} & m_8 \arrow[u, dashed] \arrow[rr, "\mu_9"'] & {} & m' \arrow[u, dashed] \\
 & & & & & & & & & M & & & & & & & & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
MM_{i-1} \arrow[r] & MM_{[i-1,\,i]} \arrow[rr, "\delta_i"] & {} \arrow[d, "G"] & MM_{[i-1,\,i]} \arrow[r] & MM_i \\
m_{i-1} \arrow[u, dashed] \arrow[r] & m_{i-1} \arrow[u, dashed] \arrow[rr, "\mu_i"'] & {} & m_i \arrow[u, dashed] \arrow[r] & m_i \arrow[u, dashed]
\end{tikzcd}$$

Figure 3.12: (left) Synthesis of migration transformation M, and (right) A generic migration step.

Automated synthesis

In the first phase, we synthesise migration transformations from difference operations. This is done automatically, by generating an instance of the default migration transformation for each difference operation corresponding to Table 3.1. The left side of Figure 3.12 shows the evolution Δ of the *Elevator* example, split up into δ_r and nine δ_i, as shown in Table 3.3. For each δ_i, a μ_i is synthesised by applying the transformation G. Conceptually, G is a higher order transformation, because it takes transformation models instead of instance models as input or output [198]. In practice, G is more efficiently implemented by using a migration template for each delta operation and instantiating this template with the parameters of the delta operation. In this modular approach, we start by using an intermediate metamodel for every step. The metamodel MM_E evolves to $MM_{E'}$, over metamodels MM_i. Using this approach the result of every migration step μ_i is syntactically limited to the delta operation δ_i instead of the full evolution. The instance model m is migrated accordingly to m'. In this case, $MM_9 = MM_{L'}$ and $m_9 = m'$. A single step is shown in a more detailed way on the right side of Figure 3.12. For every step δ_i is considered an evolution as presented in Figure 3.11, where the intermediate metamodel $MM_{[i-1,i]}$ is the merge of MM_{i-1} and MM_i. So for every step, m_{i-1} is first converted to the intermediate metamodel $MM_{[i-1,i]}$, then transformed by μ_i, and then converted to MM_i. It is clear that μ_i is an in-place transformation.

Once the default migration operation is synthesised for each δ_i, the instance models m can be migrated by executing the sequence of in-place transformations $M = \mu_i \circ \mu_{i-1} \circ ... \circ \mu_2 \circ \mu_1$ of Figure 3.12. By construction, the resulting $m' = M(m)$ will syntactically conform to $MM_{E'}$.

Manual adaptation

Technically, the first phase fulfils the requirement for co-evolution, namely ensuring that the new models conform to the new language. Syntactic migration and consistency is thereby accomplished. In the *Elevator* evolution however, there are also cases of semantic migration, thus continuity must be maintained. An example is the replacement of the UpButton and DownButton with the up_direction in the now concrete FloorButton class. In the first phase, UpButtons and DownButtons are removed as a consequence of the "eliminate class" operation, which is clearly not the intention of the evolution. Semantic migration is done during the manual adaptation phase.

In this phase, each δ_i and corresponding default μ_i are one by one presented by the language engineer. For each μ_i, the language engineer can choose from five possible actions:

— *keep* the default μ_i. If the language engineer is satisfied with the default μ_i, nothing has to be done for this step. This action is typically applied for non-breaking or breaking and resolvable

nr.	Operation	Action
δ_r	RenameClass(class=Button, name="CallButton")	*keep* (compulsory)
δ_1	RestrictProperty(prop=requests, lowsrc=0, highsrc=3, lowdst=1, highdst=1)	*edit*: μ_i = *id*
δ_2	ExtractAbstractSuperclass(name="Button", subclasses=[CallButton])	*keep*
δ_3	PullPropertyToAbstractClass(prop=pressed)	*keep*
δ_4	AddClass(name="EmergencyButton", superclass=Button, abstract=False)	*keep*
δ_5	MakeClassConcrete(class=FloorButton)	*group* and *create*
δ_6	AddObligatoryProperty(name="up_direction", owner=FloorButton, lowsrc=1, highsrc=1, type=boolean)	
δ_7	EliminateClass(class=UpButton)	
δ_8	EliminateClass(class=DownButton)	
δ_9	EliminateProperty(prop=nr)	*keep*

Table 3.4: The operational difference model Δ of the *Elevator* evolution.

changes;

— *edit* the default μ_i. The language engineer might be satisfied with the structure of the default μ_i, but might wish to alter μ_i to μ'_i. This action is typically applied for breaking and resolvable changes or breaking and unresolvable changes;

— *group* the current μ_i with following μ_{i+1}. In some cases, a number of difference operations can be grouped as one conceptual change, requiring one μ'_S (with S a sequence of consecutive indices) for two or more difference operations;

— *create* a tailored migration for the corresponding difference operation. If the language engineer has a migration transformation in mind that is completely different than the default one, he/she can create his/her own. The action is typically applied for non-breaking (if the language engineer actually wants to migrate instead of doing nothing) or breaking and unresolvable changes. Note that by first grouping and next creating, the original μ_i and μ_{i+1} are replaced by one $\mu'_{i,i+1}$ that covers both the migration of δ_i and δ_{i+1};

— *relegate* a migration to a manual intervention step in the migration transformation, by requiring user input at migration time in case of model-specific migration [86]. This action is typically applied for breaking and unresolvable changes. Of course, this needs to be avoided as much as possible, because the migration step will need to be done for each instance.

Note that μ_r has to be kept as it needs to be a purely syntactic migration.

Table 3.4 shows the result of the *Elevator* migration after the manual adaptation phase. μ_r, μ_2, μ_3, μ_4 and μ_9 are kept, μ'_1 is edited (actually, we assume that never more than three Buttons are attached to a Floor, so μ'_1 = *id*) and μ_5, μ_6, μ_7 and μ_8 have been grouped (introducing the notion of direction) and $\mu'_{5,6,7,8}$ is created manually. The resulting evolution pipeline is shown in Figure 3.13.

Figure 3.14 shows the customised migration transformation $\mu'_{5,6,7,8}$. Two rules are created. The left rule transforms UpButton instances to FloorButtons with *true* as value for up_direction (visualised by “up” in the top right corner of the FloorButton). The right rule is the equivalent for

$$\begin{tikzcd}
 & & & & & \Delta \\
MM_E \arrow[r, "\delta_r", ""'{name=Dr}] & MM_0 \arrow[r, "\delta_1"] & MM_1 \arrow[r, "\delta_2", ""'{name=D2}] & MM_2 \arrow[r, "\delta_3", ""'{name=D3}] & MM_3 \arrow[r, "\delta_4", ""'{name=D4}] & MM_4 \arrow[r, "\delta_5"] & {} \arrow[r, "\delta_6"] & {} \arrow[r, "\delta_7"] & {} \arrow[r, "\delta_8"] & MM_8 \arrow[r, "\delta_9", ""'{name=D9}] & MM_{E'} \\
m \arrow[u, dashed] \arrow[r, "\mu_r"', ""{name=Mr}] & m_0 \arrow[u, dashed] \arrow[r, "\mu'_1"'] & m_1 \arrow[u, dashed] \arrow[r, "\mu_2"', ""{name=M2}] & m_2 \arrow[u, dashed] \arrow[r, "\mu_3"', ""{name=M3}] & m_3 \arrow[u, dashed] \arrow[r, "\mu_4"', ""{name=M4}] & m_4 \arrow[u, dashed] & & {} \arrow[r, "\mu'_{5,6,7,8}"'] & {} & m_8 \arrow[u, dashed] \arrow[r, "\mu_9"', ""{name=M9}] & m' \arrow[u, dashed] \\
 & & & & & M
\arrow[from=Dr, to=Mr, "G"]
\arrow[from=D2, to=M2, "G"]
\arrow[from=D3, to=M3, "G"]
\arrow[from=D4, to=M4, "G"]
\arrow[from=D9, to=M9, "G"]
\end{tikzcd}$$

Figure 3.13: The step-wise migration after the manual adaptation phase.

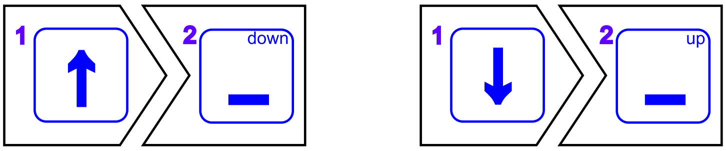


Figure 3.14: The customised migration transformation $\mu'_{5,6,7,8}$.

DownButtons.

A new problem arises when allowing the language engineer to manually create migration operations. After this phase, it cannot be guaranteed any more that m_i is conform to MM_i, as the language engineer is allowed to implement anything he/she wants in the customised migration transformations. In our approach, we offer a solution to uphold this guarantee by providing maximal control over the creation of the migration operation, while still offering full expressiveness. This control is provided by two mechanisms, a preventive mechanism and a corrective mechanism:

Restricted metamodel: as a preventive mechanism, it is only allowed to use language constructs of the corresponding difference operation(s) when editing or creating a μ'_S (with S a sequence of one or more consecutive indices, though in many cases this is just one index i, as suggested in Figure 3.13). This means that we isolate a part of the total evolution for this migration, particularly $MM_{min(S)-1}$ to $MM_{max(S)}$ (in the case of $|S| = 1$ this would be MM_{i-1} to MM_i), as intended by the step-wise migration. Again, changes of a previous evolution step δ_h with $h < min(S)$ are considered carried through, and changes any future evolution step δ_k with $k > max(S)$ are not yet considered at all. For example when creating $\mu'_{5,6,7,8}$ in Figure 3.13, the changes δ_1 to δ_4 are carried through, and change δ_9 is disregarded for now. Only for changes δ_5, δ_6, δ_7 and δ_8, will a migration transformation be created, aiding transformation modularity.

Technically, this degree of control is achieved by merging the metamodels $MM_{min(S)-1}$ and $MM_{max(S)}$ into an intermediate metamodel $MM_{[min(S)-1,max(S)]}$. This way, an in-place transformation can be created. Since in this context it is possible that a μ' is created for more than one δ, the intermediate metamodel can include more than one δ.

Checkout transformation: as a corrective mechanism, full model conformance is ensured of the partly migrated instance model to the partly evolved metamodel in the checkout transformation γ. This step is automatically achieved in our approach by applying the default migration transformations of the difference operations immediately after the customised migration step, *i.e.*, $\gamma_i \circ \mu'_i$. After all, the default migration transformation is constructed so that its output models

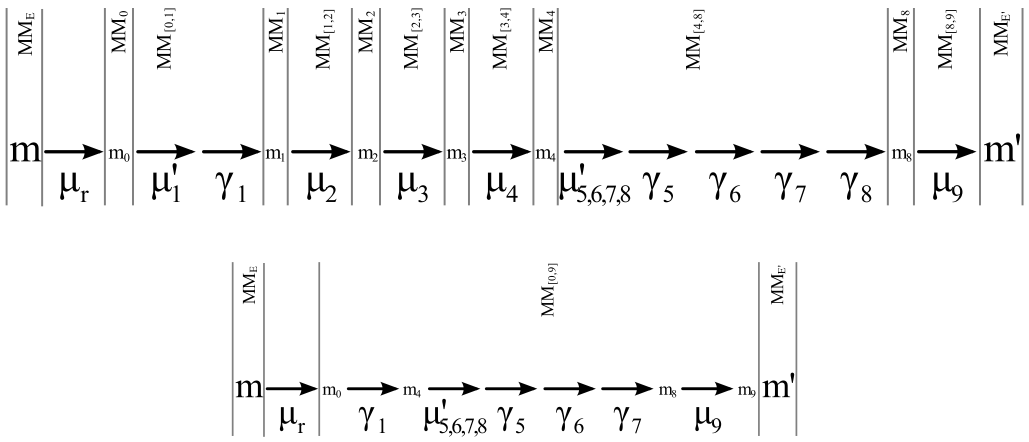


Figure 3.15: The naive execution needing a lot of conversions (top) and the optimised execution needing only three conversions (bottom).

are syntactically correct. This way, *e.g.,* instances of deleted classes that are by accident not deleted by the customised migration transformation, are deleted by the checkout transformation, thereby ensuring conformance to the partly evolved metamodel. Typically however, the language engineer has designed his/her customised migration transformation so that model conformance is already ensured. In that case, executing the checkout transformation will not change anything. The checkout transformation merely validates conformance in the general case.

M is composed of regular transformation models, and is stored as any other transformation model. Thus, future instance models conforming to the old version can be migrated later.

Execution

At first glance, the execution of the migration suite M is straightforward. On all instance models m, M is applied. More specifically a sequence of in-place transformations, like μ_i, μ'_j and γ_j, are applied in the given order. The ad hoc execution is not optimal however. As shown on the right side of Figure 3.12, in order for each of the in-place transformations to be executed, the instance model must be converted to that particular intermediate metamodel of the step. After execution, the result must be converted to the partly evolved metamodel. As previously mentioned, these conversions are trivial: a simple search/replace script on the data file of the instance model or a trivial transformation that implements a one-to-one mapping of elements can be automatically generated. However, this can cripple the execution performance of the total migration. In Figure 3.15, a conversion is needed every time a different metamodel is used (*i.e.,* a grey vertical line is crossed) throughout the execution of M. The top of Figure 3.15 represents the naive execution, requiring a multitude of conversions.

As a solution, after creation of migration transformation M, the different metamodels used in the in-place transformations are relaxed to the intermediate metamodel that spans all δ_i in Δ. In the *Elevator* example, the intermediate metamodel is shown in Figure 3.10. As mentioned, all possible instance models throughout all migration steps can be expressed in the resulting metamodel MM_Δ. Every in-place transformation's used metamodel is changed to $MM_{[0,||\Delta||]}$. Of course, this has to

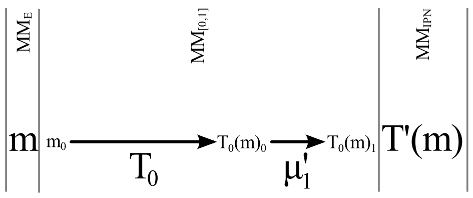


Figure 3.16: The execution of *E2PN* after image evolution.

be done only once for M instead of for all instance models. With this optimised approach, a m that needs to be migrated only has to be converted two times: (1) after retyping operations of μ_r, but before applying the in-place transformations of M (μ_r might be part of the conversion step), and (2) after applying the in-place transformations of M. In between, all artefacts use the same metamodel $MM_{[0,||\Delta||]}$, and only in-place transformations are applied. Moreover, μ_i migrations that are equal to *id* are removed. In the example, μ_i' (because we assume that no instances exist with more than three Buttons on one floor), and μ_2, μ_3, μ_4, and γ_8 (because their associated delta operations are non-breaking), can be safely removed. The bottom of Figure 3.15 represents the optimised execution. Note how γ_1 ensures consistency by removing non-compliant Buttons if our assumption of maximally three Buttons per Floor turns out to be incorrect. Also note how m_9 (conform to $MM_{[0,9]}$) is explicitly visualised to denote the difference with *m'* (conform to $MM_{E'}$).

Additionally, the absence of model-to-model transformations adds to the execution performance of the migration because after evolution, often models only change slightly, if at all. If the language engineer is confident in his/her customised migration transformation, he/she has the option to disable the execution of the checkout transformations, further improving the execution time of M.

The example did not include a *relegate* action, which imposes a manual intervention when executing the migration. In that case, a conversion right before the μ_i that was relegated and right after are kept. This way, the domain user who executes the migration transformation can only edit models within the modelling space of $MM_{[i-1,i]}$.

Note that the example presented in this section is deliberately chosen to show the different aspects of our approach. When using migration operations in practice, however, Herrmannsdörfer *et al.* [86] have shown that in industrial case studies, generated migration operations can be used for more than 99% of the total number of operations. It turns out that a majority of changes are refactorings. Note that this might be due to the complexity of other (*i.e.,* additive and subtractive) changes. In contrast to the example, editing migration operations is needed for only a limited number of cases.

3.3.3 Migration of Transformation Models

In addition to instance models, related transformation models must be migrated. A transformation may often be represented as a graph transformation model, consisting of graph rewriting rules. However, its implementation can be simpler in some cases, such as an XSLT transformation or a simple script. Transformations can be total, *i.e.,* mapping the entire domain onto the image. In many cases, however, some language constructs of the domain do not have an image, so the transformation is partial. In this case, semantic information is lost.

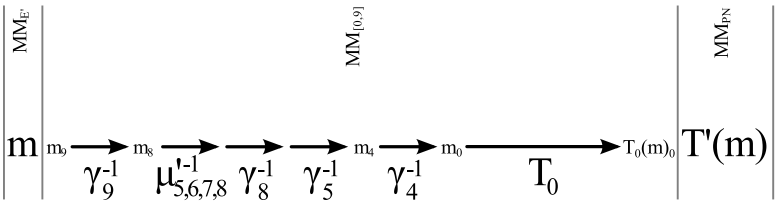


Figure 3.17: The execution of *E2PN* after domain evolution.

Suppose we apply transformation co-evolution directly to the evolution of the *Elevator* relational ecosystem of Figure 1.3. An image evolution scenario was presented in Section 3.2.6: *E2PN* co-evolves with the evolution of Petri nets. Syntactically, the co-evolved transformation would be able to transform *Elevator* models of to inhibitor Petri nets: *T'(m) = T(M(m))* as shown in Figure 3.4 (b). In that case, M consists of one migration operation μ'_1 that replaces place-antiplace patterns with one place as explained in Section 3.2.6. Note that in this co-evolution, no adaptation needs to be done to *T*. For optimisation however, *T* and *M* can be merged to a single transformation *T'*. The use of the intermediate metamodel plays a key role in the solution to this problem. As concepts of both versions of the metamodel can be modelled in the intermediate model, no information needs to be lost. The execution of the migrated transformation model is shown in Figure 3.16. Therefore, instead of editing the transformation so that it produces MM_{PN} instance models, we convert the transformation to *E2PN*$_0$ by applying rename operations (none in this case) and type conversions (as shown by the left arrow of the evolution pipeline of Figure 3.11). As a result, it produces Petri net models conform to the intermediate metamodel $MM_{[0,1]}$. Next, the migration transformation can be appended, so that the resulting *T'* produces inhibitor Petri nets, that can then be converted to MM_{IPN} by the right arrow of Figure 3.11. As now only one transformation model remains, it can be edited.

The evolution of *E* and the co-evolution of *E2PN* is, when mapped to the relational ecosystem of Figure 1.3, a case of domain evolution. As shown in Figure 3.4 (c), the co-evolved transformation *T'(m) = T(M*$^{-1}$*(m))* is syntactically able to transform *Elevator* models conform to *MME'*. In order to achieve this, M^{-1} must be defined. In that regard, it is advisable to define the reverse migration at the time of defining M, because if a DSML evolves, domain evolution will likely occur. In the example however, the information about EmergencyButtons would be lost after applying *M*$^{-1}$. This is often not desirable, as this information was deliberately added to the new version of *Elevator*. The execution of the migrated transformation model is shown in Figure 3.17. In the case of domain evolution for the semantic mapping *E2PN*, the mapping is migrated to the intermediate metamodel. Then, when a $MM_{E'}$ model must be transformed, it must be migrated "towards" the converted *E2PN*$_0$ transformation by transforming the model to m_9, conform to $MM_{[0,9]}$. Syntactically, *E2PN*$_0$ can transform m_9, because they both transform/conform to $MM_{[0,9]}$. However, as the transformation is virtually MM_0 and the model is virtually conform to MM_9, it is likely that the mapping is not total any more. For example, *EmergencyButton*s and *up_direction* attributes will not be transformed. Therefore, m_9 will pass through some chosen slots in the migration pipeline. In this example, it is desirable that FloorButtons are converted back to UpButtons and DownButtons. We decide that the semantics of EmergencyButton are not relevant for our deadlock analysis in Petri nets. Note that this judgement is not to be taken lightly, and relates to the general problem

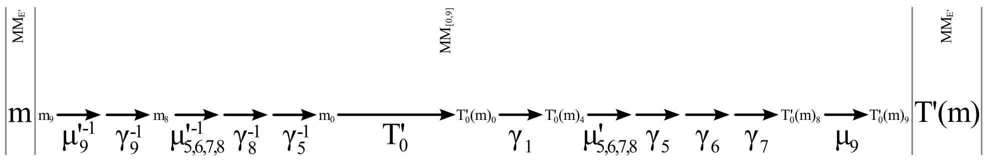


Figure 3.18: The execution of $[[.]]_{OS}$ after domain and image evolution.

of abstraction. Consequently, only the inverse migration operations of δ_5 to δ_8 of Figure 3.15 are applied on the instance model, and checkout transformations for the inverse of δ_9 and δ_4. More specifically, γ_9^{-1} introduces the nr attribute in each Floor (note that in this context a default value of 0 can be given as it is not used in the *E2PN* transformation), and γ_4^{-1} removes all instances of EmergencyButton. The other inverse delta operations are non-breaking. As a result, the migrated transformation *E2PN* produces useful models again. Similar to image evolution, the migrated transformation can be edited for optimisation.

Finally, consider the evolution of *E* and the co-evolution of $[[.]]_{OS}$ in the *Elevator* relational ecosystem of Figure 1.3. An implementation of this co-evolution scenario is shown in Appendix C. In the endogenous $[[.]]_{OS}$ transformation, both image and domain evolves, so both M and M^{-1} will have to be employed as shown in Figure 3.18. Since $[[.]]_{OS}$ makes use of the nr attribute of Floors, γ_9^{-1} will not suffice and a $\mu_9'^{-1}$ must be constructed to set nr values for each floor. The transformation $[[.]]_{OS}$ can be reused, and has to be adapted only to add semantic information about EmergencyButtons. Therefore, γ_4^{-1}, which simply removes EmergencyButtons, is removed from the pipeline and *E2PN* is adapted (denoted as T_0'). In conclusion, the migration transformation for instances can be reused as a structured starting point for transformation co-evolution. The unavoidable manual adaptation of the co-evolving transformation is kept to its essence, *i.e.,* the inclusion of changed semantics.

3.4 A Framework for Modelling Language Evolution

In this section, a top-down approach for a framework for evolution of modelling languages is presented. The concepts of previous sections will thus be mapped out in one framework. The main requirement of this framework is that it must be complete, which means that it must be able to handle every possible case of language evolution. Furthermore, it must present a solution that maximises automation. This means that user intervention must be limited, and directed by the framework. We can distinguish three levels of automation, ordered by the level of user intervention required:

— full automation. This highest form of automation can be implemented in the framework itself. If used, it does not require any manual intervention;

— evolution-specific automation. This form of automation is achieved at the level of evolution itself, by the language engineer. As a result, the corresponding co-evolutions can be performed automatically;

— no automation. If manual intervention is needed at the level of the co-evolving instances, by the domain user, there is no automation.

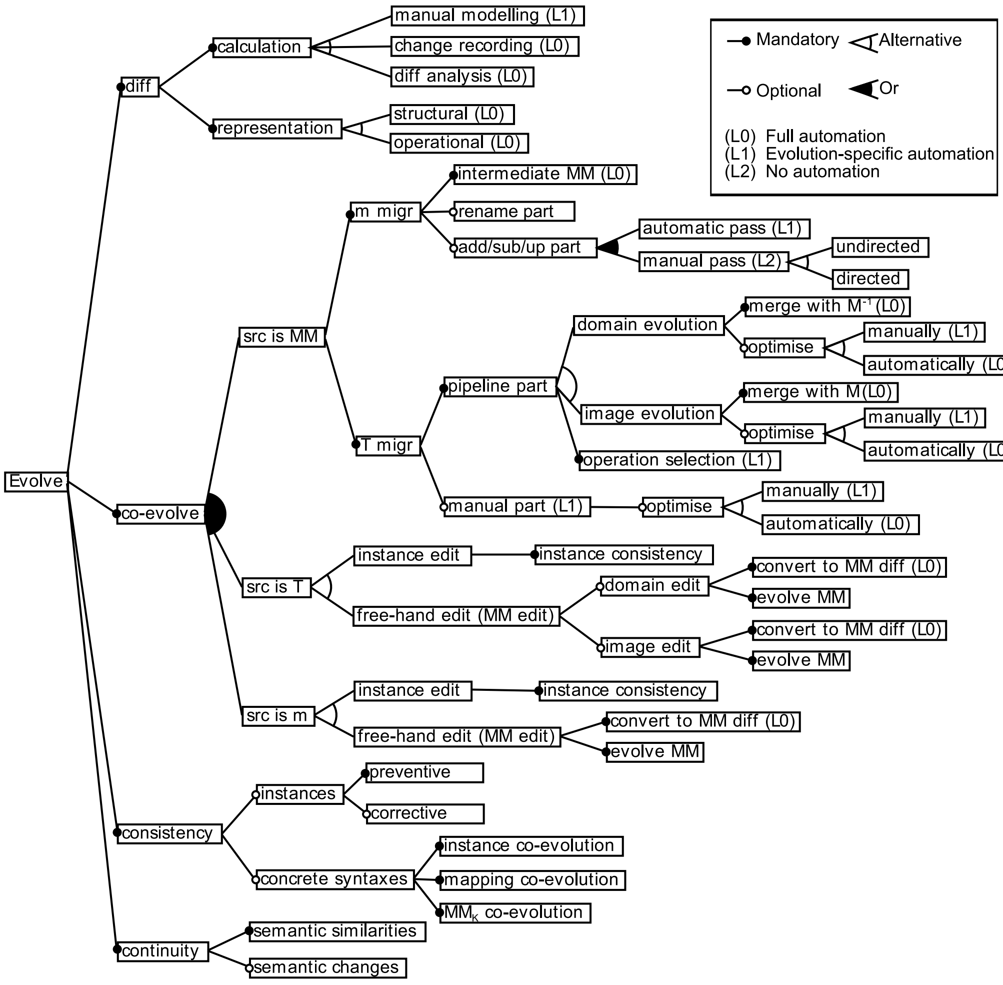


Figure 3.19: Feature diagram of the framework for evolution of a modelling language

Although full automation is desirable, the other classes of automation are sometimes necessary as shown in the previous section.

Figure 3.19 shows a feature diagram [93] of the framework that is explained in this section. Feature diagrams are a model for product families, and they model how different products can be composed. In our context, we use it to describe a *family* of migrations, *i.e.,* the diagram models all possibilities for the evolution of modelling languages. The feature diagram shows what possibilities have to be taken into account for the creation of a framework. Afterwards, an algorithm is provided. Throughout the remainder of this section, the feature diagram is explained, and features from the feature diagram are referred to in italic font. In order to support evolution (*Evolve* in the feature diagram), the framework must support:

— model differencing (*diff*);

— co-evolution (*co-evolve*) of instance models or related transformation models. Depending on the context, this can originate from metamodel changes (*src is MM*) or changes of another artifact (*src is T* and *src is m*). More than one possibility emerges here: an evolving artifact can be an instance model in one mapping to the basic schema, a metamodel causing domain evolution in another mapping to the basic schema, a metamodel causing image evolution in yet another mapping to the basic schema;

— *consistency* of the system;

— *continuity* of the system;

These features are explained further in this section.

3.4.1 Difference between Models

The evolution of an artifact is modelled explicitly in a delta model, in order to derive a migration transformation (semi-)automatically. Therefore, the difference between two versions of a model needs to be represented (*representation*). As discussed in Section 3.5.1, there are two kinds of representations: *operational*, where difference is modelled as a series of edit operations, and *structural*, where difference is modelled using language elements (represented explicitly in a metamodel).

Apart from representation, the *calculation* of the difference model is an essential part of the framework. The difference models can be modelled manually (*manual modelling*), resulting in evolution-specific automation. Full automation can be achieved when the difference is calculated from the two versions of the metamodel (*diff analysis*), or when the changes are recorded at the moment they are applied (*change recording*). Change recording is very accurate, and does not require a complex algorithm to be implemented. However, it is dependent on that particular framework and difficult to visualise. The differences in change recording are conveniently represented as an operational model.

3.4.2 Evolution of Metamodels and Co-evolution

When following the migration approach of Section 3.3.2, a migration model (*m migr*) involves creating an intermediate metamodel, a rename part δ_r and an additive/subtractive/updative part. As stated in Section 3.2.2, the intermediate metamodel (*intermediate MM*) describes all concepts of both metamodel versions, and can be generated fully automatically. The *rename part*, consisting of all rename delta operations, can be automatically derived from the difference model. As rename operations are not subject to the projection problem, migration operations and their inverse (as necessary when applying transformation co-evolution) can be generated automatically. The additive/subtractive/updative part (*add/sub/up part*), consisting of all delta operations other than renaming, might need manual adaptation, so possibly no automation can be applied to some part of the co-evolution. As a result, co-evolution is performed using *automatic pass*(es) and *manual pass*(es). The manual pass can require *undirected* ad hoc adaptation, or can be *directed* by the framework, which points out where manual intervention must be done in the co-evolving artifact, as explained in Section 3.3.2.

In our framework we divide transformation co-evolution (*T migr* in Figure 3.19) in two parts: a pipeline part and a manual part. The first part (*pipeline part*) can be automatically executed using a combination of the evolution pipeline and the principles of *image evolution* and *domain evolution* as explained in Section 3.3.3. This means that the transformation is merged with M (*merge with M*) or M^{-1} (*merge with* M^{-1}). This process can be optimised (*optimise*) *manually* or *automatically*. As noted, the simplistic approach of image- and domain evolution is not directly applicable to additive or subtractive changes. Migration of a transformation model possibly requires a *manual part*, typically used for the addition of new semantics. If an evolution comprises a lot of semantic changes, the amount of effort that has to be put in this manual part will be greater. Note that, when a series of evolutions are performed, the intermediate metamodel might describe a larger modelling space, as concepts of all evolutions are incorporated. On top of this, the number of slots in the pipeline will grow, as each evolution change is described by these migration operations. Therefore, it is desirable to "freeze" the evolved system at some point, and *optimise* the migrated transformations. This can be done either *manually* or *automatically*. An automated approach is presented in [13]. From then on, the system can again be considered version 0.

If an instance model changes (*src is m*) or a transformation model changes (*src is T*), there are two possibilities: either there is no impact on a metamodel (in case of regular *instance edit*) or there are metamodel changes (in case of *bottom-up edit*). For the former, some transformations may have to be executed again in order to maintain consistency among the model instances (*instance consistency*). The latter is more complex and occurs when a tool supports bottom-up editing [141, 40]. In bottom-up editing, the modeller can immediately edit an instance or concrete syntax in order to change a language. The conversion of the m/T difference to a metamodel difference (*convert to MM diff*) is supported by the bottom-up editor. Whenever the editor applies a change to an instance model in the language, the change is generalised and applied to a (possibly implicit) metamodel. Note that this is typically not done automatically, but rather, suggestions are given. Ultimately, the modeller is presented a metamodel editor to allow manual metamodel edits. Freehand editing can only be done if the other model is closely related to the original model, by a homomorphic mapping. Homomorphic relations are in this context metamodel/instance or abstract syntax/concrete syntax relations. When the metamodel changes (*evolve MM*), a new evolution is triggered. Note that for a transformation model change, the domain metamodel might be subject to change (*domain edit*) as well as the image metamodel (*image edit*).

3.4.3 Consistency and Continuity

When a language evolves and instance models and transformation models co-evolve, the system might have become inconsistent (*consistency*). In many cases, it is desirable to solve this by executing respective transformations so that the model instances are consistent again (*instances*). In Section 3.3.2 *preventive* (using small steps with each their own intermediate metamodel) and *corrective* (optionally using checkout transformations) mechanisms are presented. After the necessary transformations have been executed, the system is consistent again. We assume that all evolutions and co-evolutions are done correctly, *i.e.,* they can result in a consistent system.

In addition to the consistency of instance models, we also have to take consistency between abstract syntax and concrete syntaxes into account (*concrete syntaxes*). Due to the homomorphic nature of the concrete syntax mapping and the tight coupling, this problem is usually simpler than other co-evolution problems, as the concrete syntax mapping mainly consists of only simple one-to-one transformation rules. When changing abstract syntax, the metamodels of the corresponding concrete syntaxes must co-evolve. This requires migration of the metamodel of each concrete syntax (*MM_κ co-evolution*). Also, κ and π themselves have to co-evolve (*mapping co-evolution*), and the concrete syntax instances can be migrated as regular instances of a metamodel (*instance co-evolution*). The other way around, in a bottom-up modelling tool, when a concrete syntax structurally evolves it requires co-evolution of its abstract syntax. This will in turn trigger co-evolution of the other concrete syntaxes, if available. In that case, co-evolution also happens for instance models and the transformation model (*i.e.,* concrete syntax mapping).

In order to perform a meaningful evolution, it must be *continuous*. This means that the system is consistent and semantically equal to its previous versions (*semantic similarities*), with intended changes taken into account (*semantic changes*). This can in principle be checked by executing the semantic mapping and comparing the properties on these models by analysing their results. The parts of the evolution that are semantic are isolated, so that they can be incorporated in a controlled way.

3.4.4 Algorithm

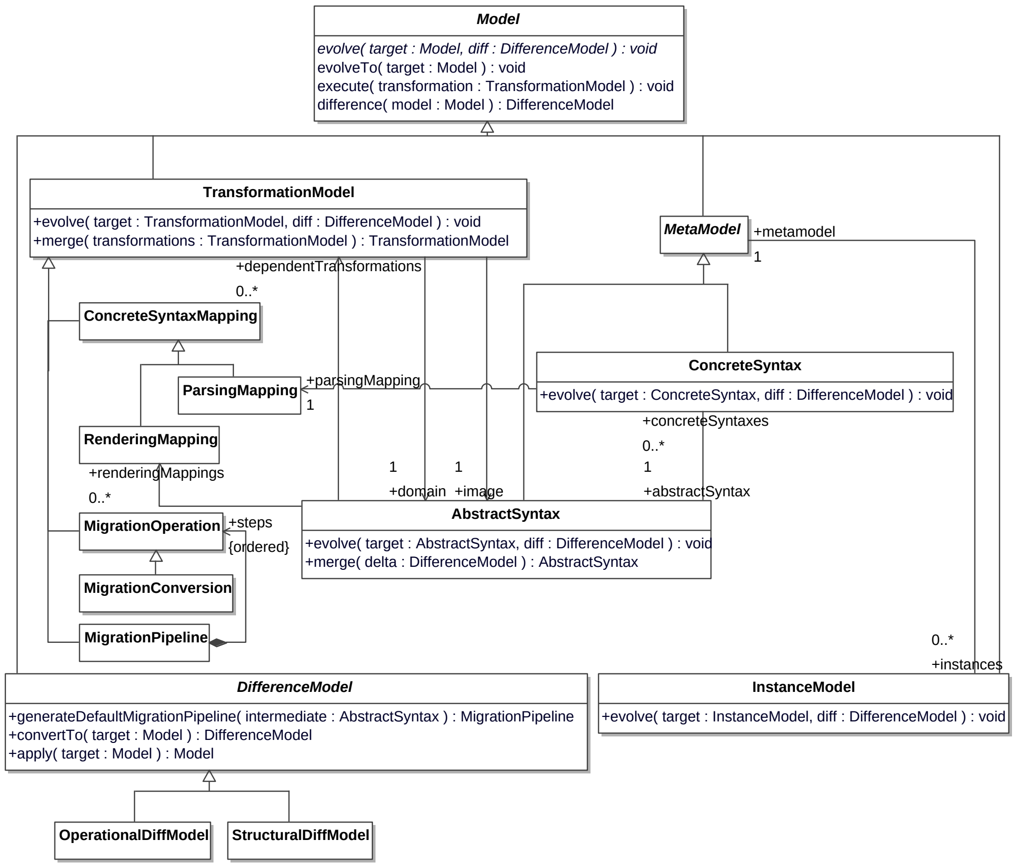

Model
evolve(target : Model, diff : DifferenceModel) : void
evolveTo(target : Model) : void
execute(transformation : TransformationModel) : void
difference(model : Model) : DifferenceModel
TransformationModel
+evolve(target : TransformationModel, diff : DifferenceModel) : void
+merge(transformations : TransformationModel) : TransformationModel
+dependentTransformations
0..*
MetaModel
+metamodel
1
ConcreteSyntaxMapping
ParsingMapping
+parsingMapping
1
ConcreteSyntax
+evolve(target : ConcreteSyntax, diff : DifferenceModel) : void
+concreteSyntaxes
0..*
RenderingMapping
+renderingMappings
0..*
1
+domain
1
+image
1
+abstractSyntax
MigrationOperation
+steps
{ordered}
AbstractSyntax
+evolve(target : AbstractSyntax, diff : DifferenceModel) : void
+merge(delta : DifferenceModel) : AbstractSyntax
MigrationConversion
MigrationPipeline
0..*
+instances
DifferenceModel
+generateDefaultMigrationPipeline(intermediate : AbstractSyntax) : MigrationPipeline
+convertTo(target : Model) : DifferenceModel
+apply(target : Model) : Model
InstanceModel
+evolve(target : InstanceModel, diff : DifferenceModel) : void
OperationalDiffModel
StructuralDiffModel

Figure 3.20: Class Diagram of the different kinds of model artifacts in the framework

In this section, we describe an algorithm as the backbone of the proposed framework. In Figure 3.20 a class diagram that is used by the algorithm is shown as an illustration. As this research is in the context of MDE, it is not surprising that the root class is *Model*. Its subclasses are:

— *TransformationModel*, that can transform models to other models. A *TransformationModel* can be a *ConcreteSyntaxMapping* (which can be a *RenderingMapping* or a *ParsingMapping*), a *MigrationOperation* or a *MigrationPipeline* which is an ordered set of *MigrationOperation*s. A *MigrationOperation* can be a *ConversionOperation*;

— *InstanceModel*, that conforms to a metamodel;

— *MetaModel*, that must be either an *AbstractSyntax* or a *ConcreteSyntax*;

— *DifferenceModel*, that is a model that describes the difference between two models. A *DifferenceModel* is either an *OperationalDiffModel* or a *StructuralDiffModel*.

Further in this section, an algorithm is shown for the *evolve* and *evolveTo* methods. These methods offer a complete framework for language evolution in MDE, that can be used to tackle all types of evolution. In the algorithms, methods that are shown in Figure 3.20 are used. Some of the used methods (*i.e.,* the association ends) are simple *getters*, more complex methods were discussed previously in the chapter. The methods can be implemented as one wishes. This means that an incremental implementation of the framework is possible using the algorithm, where initially the implemented methods require a lot of manual work (*e.g.,* the manual modelling of the delta model), but gradually they are automated (*e.g.,* by "plugging in" the DSMDiff algorithm [120] for calculating the delta model). The used methods are:

— *Model::execute*: executes a transformation with the model as input. The target is known by the transformation;

— *Model::difference*: calculates the difference of two models and returns a difference model (see Section 3.5.1);

— *TransformationModel::merge*: merges two transformation models. This can be done trivially by appending them (see Section 3.2.9 and Section 3.3.3);

— *TransformationModel::domain*: returns the metamodel of the domain of the transformation;

— *TransformationModel::image*: returns the metamodel of the image of the transformation;

— *MigrationPipeline::steps*: returns the sequence of migration steps that form the migration pipeline. The conversion steps (*MigrationConversion* instances) from the old version to the intermediate version, and from the intermediate version to the new version, are included (see Section 3.3.2);

— *DifferenceModel::generateDefaultMigrationPipeline*: generates the default migration operations following Table 3.1;

— *DifferenceModel::convertTo*: converts the difference model of an original model to another model;

— *DifferenceModel::apply*: applies the difference model to a model, so that the evolved model is obtained;

— *MetaModel::instances*: returns all available models that conform to this metamodel;

— *ConcreteSyntax::parsingMapping*: returns the parsing transformation;

— *ConcreteSyntax::abstractSyntax*: returns the corresponding abstract syntax;

— *AbstractSyntax::merge*: generates the intermediate metamodel for a given difference model (see Section 3.3.1);

— *AbstractSyntax::dependentTransformations*: returns all transformations that use this metamodel (for its input language and/or output language). It is assumed that these transformations only have one input language and one output language. If in reality a transformation has two output languages, than this transformation is returned twice, once for the first output language, and once for the second one. This assumption is introduced to improve the simplicity of the algorithms further in this section;

— *AbstractSyntax::concreteSyntaxes*: returns the corresponding concrete syntaxes;

— *AbstractSyntax::renderingMappings*: returns the rendering transformation;

— *InstanceModel::metamodel*: returns the metamodel of this model.

Figure 3.21 shows the method in pseudo code for the default case of abstract syntax metamodel evolution. It executes the consequences of a metamodel, according to the new version of this metamodel and the difference model. First, the intermediate metamodel and the migration pipeline are created (lines 2-4). The *INPUT()* method denotes manual adaptation by the language engineer. Then, instance models can be migrated (lines 5-9) by iterating over the migration operations and the instance models. Next, the relevant transformations are migrated (lines 10-22) by first customising the pipeline and, if necessary, the transformation itself (lines 11-12). Then, a distinction is made between domain and image evolution (lines 13-21), and the transformation and pipeline are merged accordingly. Note that the pipeline is traversed in the opposite way for domain evolution, so that the inverse migration steps are used. In case of domain evolution, the transformation has to be executed again on all instance models (lines 18-20). Finally, the concrete syntaxes are migrated (lines 23-36). Hereby, the homomorphic relationship between abstract syntax and concrete syntax is used extensively. The concrete syntax metamodel has to be migrated (line 24), as well as the rendering mapping (lines 25-30), the parsing mapping (lines 31-32) and the concrete syntax instances, which can be migrated by executing the migrated rendering mapping (lines 33-35).

Figure 3.22 shows the method for concrete syntax evolution in the more complex bottom-up editing. The difference is converted to the corresponding concrete syntax metamodel, and the algorithm of Figure 3.21 is called. As a result, the metamodel evolves and will migrate related artifacts as described above.

Figure 3.23 shows the method for transformation evolution in the more complex bottom-up editing. Both image and domain can change through the changes of a transformation model. The difference model is converted to the domain metamodel, and the metamodel is evolved if there was a difference (lines 2-6). Analogously, the image is evolved (lines 7-11).

Figure 3.24 shows the method for instance model evolution in the more complex bottom-up editing. Analogously to Figure 3.23, the difference model is converted to the metamodel and the metamodel is evolved.

Figure 3.25 shows the method for evolution without the knowledge of the difference model.

```
function void AbstractSyntax::evolve(AbstractSyntax target, DifferenceModel diff) do // overridden from Model
    MetaModel im = this.merge(diff) // create intermediate metamodel
    MigrationPipeline pipe = diff.generateDefaultMigrationPipeline(im) // create migration pipeline
    pipe = INPUT(pipe) // manually adapt, combine and optimise operations in pipeline
    for all (MigrationOperation mm in pipe.steps()) do // migrate instance models
        for all (InstanceModel i in this.instances()) do
            i.execute(mm) // automatic or manual execution
        end for
    end for
    for all (TransformationModel t in this.dependentTransformations()) do // migrate transformations
        MigrationPipeline custompipe = INPUT(pipe) // manually adapt, combine and optimise operations in pipeline
        for each t
        t = INPUT(t) // the manual pass: semantic changes applied to the transformation model itself
        if (this == t.image()) then // image evolution
            t = t.merge(custompipe) // the transformation, followed by migration from intermediate model to new version
        end if
        if (this == t.domain()) then // domain evolution
            t = custompipe.merge(t) // migration from new version to intermediate model, followed by the transformation
            for all (InstanceModel i in this.instances()) do // execute the transformation again
                i.execute(t)
            end for
        end if
    end for
    for all (ConcreteSyntax c in this.concreteSyntaxes()) do // migrate concrete syntaxes
        c = diff.convertTo(c).apply(c) // use homomorphic relation to convert diff and apply
        for all (RenderingMapping kappa in this.renderingMappings()) do // find rendering mapping for this concrete
        syntax to migrate
            if (kappa.image() == c) then
                kappa = diff.convertTo(kappa).apply(kappa) // use homomorphic relation to convert diff and apply
                break() // stop looking when it is found
            end if
        end for
        ParsingMapping kappaInv = c.parsingMapping() // get parsing mapping to migrate too
        kappaInv = diff.convertTo(kappaInv).apply(kappaInv) // use homomorphic relation to convert diff and apply
        for all (InstanceModel i in this.instances()) do // synchronise concrete syntax models with abstract syntax model
            i.execute(kappa)
        end for
    end for
end function
```

Figure 3.21: The *evolve* method for *AbstractSyntax*.

```
function void ConcreteSyntax::evolve(ConcreteSyntax target, DifferenceModel diff) do // overridden from Model,
assumes bottom-up modelling
    AbstractSyntax as = this.abstractSyntax() // get the corresponding abstract syntax that will evolve
    DifferenceModel diffAs = diff.convertTo(as) // use homomorphic relation to convert diff
    AbstractSyntax asEvolved = diffAs.apply(as) // apply diff to obtain new version of metamodel
    as.evolve(asEvolved, diffAs) // evolve the abstract syntax
end function
```

Figure 3.22: The *evolve* method for *ConcreteSyntax*.

The previous algorithms are in principle only usable for tools that record differences and obtain the difference model automatically. Therefore, the method of Figure 3.25 is included, where the difference is calculated by a difference algorithm, and the model is evolved, calling any of the other algorithms.

```
function void TransformationModel::evolve(TransformationModel target, DifferenceModel diff) do // overridden from Model, assumes bottom-up modelling
    AbstractSyntax domain = this.domain() // get the corresponding domain metamodel that will evolve
    DifferenceModel diffDom = diff.convertTo(domain) // use homomorphic relation to convert diff
    if not (diffDom is Null) then // check whether there was a change for the domain metamodel
        domain.evolve(diffDom.apply(domain), diffDom) // evolve the domain metamodel
    end if
    AbstractSyntax image = this.image() // get the corresponding image metamodel that will evolve
    DifferenceModel diffImg = diff.convertTo(this.image()) // use homomorphic relation to convert diff
    if not (diffImg is Null) then // check whether there was a change for the image metamodel
        image.evolve(diffImg.apply(image), diffImg) // evolve the image metamodel
    end if
end function
```

Figure 3.23: The *evolve* method for *TransformationModel*.

```
function void InstanceModel::evolve(InstanceModel target, DifferenceModel diff) do // overridden from Model, assumes bottom-up modelling
    MetaModel mm = this.metamodel() // get the corresponding metamodel that will evolve
    DifferenceModel diffmm = diff.convertTo(mm) // use homomorphic relation to convert diff
    mm.evolve(diffmm.apply(mm), diffmm) // evolve the metamodel
end function
```

Figure 3.24: The *evolve* method for *InstanceModel*.

```
function void Model::evolveTo(Model target) do // main method
    DifferenceModel diff = this.difference(target) // create difference model using a difference algorithm
    this.evolve(target, diff) // now that the difference is calculated, evolve the model
end function
```

Figure 3.25: The *evolveTo* method for *Model*.

3.4.5 Prerequisites

We assume that an average MDE tool supports metamodelling, transformation modelling and transformation execution. Following the discussion in this section, however, the proposed approach relies on a few other techniques. As these techniques are not yet featured in the average MDE tool, *maximally* automating all forms of evolution depends on the implementation of the following:

— *metamodel transformation*: the application of delta models and the generation of intermediate metamodels requires the possibility to transform metamodels. To allow this, metamodels should be modelled explicitly in a modelling language so they can adopt the role of an instance model for the metamodel transformation;

— *higher order transformation*: the automatic generation of migration transformations out of delta models requires support for higher order transformations, which are transformations that take other transformations as input and/or output. In order to support higher order transformations, the transformation language must be modelled explicitly (*i.e.,* the metamodel is not available). Several other uses for higher order transformation, inside and out of the context of evolution, are discussed in [31, 137, 147]. It is widely accepted that higher order transformations are a valuable feature in any MDE tool;

— *model differencing*: in order to support automated evolution on an industrial level, it must be possible to generate delta models out of two versions of a model. Moreover, it is desirable that the activity of metamodelling does not have to change in order to support automated evolution. Difference tools such as DSMDiff [120] and UMLDiff [215] are useful as they detect differences retrospectively (see also Section 3.5.1);

— *transformation inverting*: in order to automatically co-evolve a transformation model in domain evolution, the inverse of the migration transformation is needed. This can be implicitly featured by providing the possibility to implement bi-directional transformations using Triple Graph Grammars (TGGs) [184]. However, in that case, one is restricted to the use of bi-directional transformation with triple graph grammars. It remains an open question whether TGGs are expressive enough to obtain the inverse of the migration transformation (which may delete elements). The other option is to (partially) automate the inversion of transformation models;

— *representation of semantics*: as not only the syntax but also the semantics of a modelling language evolves, there must be a way to represent these semantic changes. The role of semantics is very apparent when reasoning about automation of language evolution. A more precise means to reason about semantics preservation is needed. Therefore, it is interesting to look more into the properties of a model and into ontologies [12]. Also, the intents catalogue presented by [124] can serve as an aid for more directed evolution. It remains to be determined whether all kinds of semantic changes can be represented uniformly, when applying this (ongoing) research to DSML evolution.

— *merging of transformations*: in order to optimise the sequences of transformations, they are merged. In this sense, automatic merging of transformations can be of great convenience. This can be simply done by chaining them, but the result can also be optimised.

The development of each of these techniques is a research topic in its own right. If all of these techniques are supported by an MDE tool of choice, a framework for evolution can provide maximal automation using the algorithm of Section 3.4.4. Nevertheless, the framework is still valuable even if the prerequisites do not hold.

3.5 Related Work

In this section, work related to evolution is presented and some useful concepts are introduced. This mainly covers the related topics of model differencing and model co-evolution. The research in this chapter was conducted in 2009–2011, but subsequent related work shows that the presented approach remains largely valid.

3.5.1 Model Differencing

A considerable amount of research has been done in the area of model differencing. Existing approaches typically rely on the abstract syntax graphs of the two models to compare, and traverse both graphs in parallel. Nodes in the graphs are matched by matching unique identifiers [1, 154], or by a number of heuristics [120, 215, 99]. This research recently culminated in the implementation of a comparison tool for the EMF framework called EMFCompare[2]. Van den Brand *et al.* describe methods for assessing the quality of tools for model comparison [203], and applied their approach

[2]`http://www.eclipse.org/emf/compare/`

to EMFCompare and their own tool RCVDiff. Specifically for metamodels, Liu *et al.* present the MMDiff tool, which supports low-level differences between metamodels [122]. In the context of UML, high level operations can be used in order not to lose the intent of the changes [220]. In this work the operations are similar to our delta operations. A metamodel-specific, highly configurable matching algorithm is presented in [204]. The approach is based on an extended metalanguage (metametamodel) based on MOF that includes an explicit conformance relationship between metamodels and models. Könemann on the contrary, presents simple idioms to capture the intention of model changes [106].

In difference representation, approaches are defined for an operational representation [1, 86], possibly as a model transformation [111]. In structural (or state-based) representations, either the model is coloured [120, 154, 183, 215] or a designated delta model is created which can be used by modelling tools as yet another model in an appropriate language [30, 187, 112]. Van den Brand *et al.* visualise the difference model as a combination of polymetric views for representing the overview (suitable for zooming and filtering) and a generic visualisation framework for showing the details of the difference model [205]. Di Rocco *et al.* step away from studying the difference between just two models, and provide an approach to assess the impact of a change on related modelling artifacts [50]. The impact can be visualised by keeping track of dependencies related to the changes, by means of a traceability model. Interestingly, the authors also study the impact of evolution in an ecosystem, much like our concept of a relational ecosystem.

Porres describes the state of the art in model differencing [163], and identifies the relationship between an original model M_1, an evolved model M_2, and the difference Δ as $M_2 - M_1 = \Delta$ and $M_1 + \Delta = M_2$. In our approach, we circumvent the problem of differencing by starting from the second equation. Porres describes a very similar merge operation as the one we present to calculate the intermediate metamodel.

A general survey on model matching techniques can be found in [190], including the application of reviewed approaches and a comparison of the kind of models that approaches are intended to work with. An alternative to differencing is change recording, which is presented by Herrmansdörfer *et al.* in the context of EMF [87].

Model versioning goes further than differencing, as it presents a multi-user view on model changes. As a result, conflicts need to be detected, and resolved. Mens presents a classification of resolutions for merging code in textual merging, syntactic merging and semantic merging [130]. Although, because of the graph-like nature of models textual merging is not considered a valid technique in model versioning, syntactic and semantic merging (of either static semantic conflicts or behavioural conflicts) and ditto differences are applicable to models too. When extrapolating this classification to our work, our concept of consistency aims for syntactic validity similarly to syntactic merging or semantic merging of static semantic conflicts, and our concept of continuity aims for “intended behaviour” similarly to semantic merging of behavioural conflicts. Similar to model differencing, structural (called state-based), operational (called change-based) and transformation-based (called operation-based) differences are identified. For graph-based structures, Täntzer *et al.* distinguish between operation-based conflicts and state-based conflicts and present merge actions for both [195]. Interestingly, the approach orders changes so that first the additive changes are applied, and later the subtractive changes, to keep as much information as possible. The authors present an implementation in EMF. The link between model versioning and metamodel evolution is presented in [29], where metamodel and model changes are merged. Merging is performed on the difference models and can be stored as such in the repository. Challenges in model versioning are described in [2], also providing an overview. A more recent overview was presented by Brosch *et al.* [95, 27].

3.5.2 Co-Evolution of Modelling Artifacts

The co-evolution (with evolution of their metamodel) of models has become an important research topic. This research is inspired by the way the problem of language evolution is identified or dealt with in other domains. In grammar evolution [102, 113, 161], the type/instance relation is analogous to the metamodel element/instance relation. This type/instance relation is also present in programming languages. In database schema evolution, database tables have to be migrated after a change in the database schema [9, 119, 165] (cf. [175] for a survey). In format evolution, formally specified documents (*e.g.,* XML documents) must be migrated when their format (*e.g.,* specified in a DTD) changes [114, 192]. However, in modelling language evolution, the changes to model semantics add a new "twist" to the evolution problems faced in these domains.

Many approaches represent model instance co-evolution as a model transformation [31, 77, 85, 86, 88, 187, 209, 212, 219]. Gruschko *et al.* write this transformation manually using the Epsilon Transformation Language (ETL) [77]. They provide an automated copy operator that allows the modeller to refrain from tediously writing ETL code for the model elements that do not change. This algorithm is implemented in model migration framework Flock [176]. This default rule generates target elements based on name equivalences between the metamodel versions. By this, only transformation rules have to be manually developed for metamodel elements which have been modified during metamodel evolution.

Sprinkle *et al.* [187] considered co-evolution of models by using changed semantics to design co-evolution transformations. This differs from a syntactically driven approach that uses the metamodel deltas. In that work as well as in [186], the authors proposed that syntactic co-evolution (where the importance is only to load, but not interpret, the models) is feasible automatically, but it seems to be impractical for semantic evolution. In the general case of semantic evolution concerns, semantics-preserving transformations must be developed by language engineers manually, based on their understanding of the semantic intent of the original models. However, for specific cases, semantically-preserving co-evolution transformations are possible. In this work, we are following this distinction.

There are several approaches for co-evolution which are based on model-to-model transformations. Garces *et al.* [67] proposed a set of heuristics to automatically compute differences between two metamodel versions in order to adapt models. The computed differences are stored in a so-called matching model, acting as input for a higher-order transformation (HOT), producing a migration transformation. This is quite similar to our approach. The approach is implemented in a prototype called AML and applied to evolution of the Java Netbeans design. Cicchetti *et al.* [31] also presented a similar approach, *i.e.,* the approach is again based on metamodel differences acting as input for a HOT. Van den Brand *et al.* [206] presented a full approach by first calculating the differences of metamodels. These metamodels are instances of a metametamodel, so model difference techniques can be applied. The resulting difference operations are used as input for a generic transformation that migrates instances, instead of using a HOT to generate a migration transformation. This corresponds to our implementation in *metaDepth* presented in Appendix C.

In [212], Wachsmuth proposed to combine ideas from object-oriented refactoring and grammar adaptation to provide the basis for automatic (meta)model evolution. In this respect, metamodel relations are defined based on model-to-model transformations, building the basis for the definition of semantics preservation and instance preservation. Moreover, a set of transformations are proposed, which are accordingly classified into refactoring, construction, and destruction transformations. In [150, 117] the Model Change Language (MCL) is introduced. MCL is a declarative and graphical language supporting a set of co-evolution idioms and conservative copying. The language engineer defines relationships between elements of the metamodel versions. There are different kinds of relationships starting from simple one-to-one mappings between classes to more complex mappings for typing objects to new subclasses or for changing the containment hierarchy. More complex co-evolution rules have to be defined in the context of the defined mappings as constraints and actions in terms of C++ code. For unchanged parts of the metamodel no mappings have to be defined due to the usage of name equivalences similar to Gruschko *et al.* The provided idioms only cover syntactic co-evolution concerns but not semantic concerns. Most of the mentioned model-to-model-based approaches intend to shield a language engineer from creating standard copy rules by providing matching techniques or conservative copying techniques. However, the non-automatically derivable parts have to be manually defined which seems to be more challenging for the language engineer compared to using in-place transformations. This is due to the fact that the language engineer has to reason about what elements look like in the source model, how elements are represented in the target model, and how they are transformed by analysing the trace information enforcing the user to work with three models. Di Rocco *et al.* specifically focus on user intervention [51]. They provide an interactive tool where the user has to make simple choices at given times, *e.g.,* selecting which instances have to be removed if an association cardinality is restricted.

Most approaches define some specific operations as building blocks for evolution, similar to the operational representation of model differences. Such operations typically include "create class", "restrict multiplicity on association" and "rename attribute" and are related to object-oriented refactoring patterns. Migration operations can be generated from sequences of delta operations. It is important that any possible evolution can be modelled. There is a general consensus that the proposed sets of delta operations do not suffice. Herrmannsdörfer *et al.* try to solve this problem by repeatedly extending their list of delta operations [86]. The approach is implemented in COPE, a tool for specifying the coupled evolution of metamodels and models. The co-evolution of metamodels and corresponding models is realised by a set of so-called coupled operations, composing a whole co-evolution problem of modular in-place transformations. Interestingly, they classify migration operations with respect to their reuse: operations can be reused across different metamodels, only across models of the same metamodel, or not at all. Expressiveness is ensured by including custom coupled operations, but these must be expressed manually by primitives embedded into the language Groovy. The authors observe that such unreusable operations, which they call model-specific coupled changes, are only needed in less than 1% of the migrations in their industrial use cases, when using an elaborated list of operations. Like COPE, we utilise an incremental evolution approach, refraining from a single evolution process. However, our incremental process is supported by computing intermediate merged metamodels, thus we allow to model the migration of models by ensuring all metamodel constraints. Although the automated synthesis of the intermediate metamodels gives up some control, it provides an ability to verify well-formedness, and differs from the metamodel-independent representation of models used in COPE, which lacks the possibility for intermediate validation.

Although a less popular research topic than model co-evolution, the problem of transformation co-evolution already described by Zhang *et al.* in 2004 [219]. Similar problems for grammar transformations are addressed by Zaytsev [218]. In his work, a new model for grammar transformation is presented to make them more adaptable. Similar commuting diagrams are identified, and the concepts of image and domain evolution are reused. The approach includes explicit support for manual intervention.

Levendovszky *et al.* state that transformation co-evolution is more complex, thus less automatable, than instance model co-evolution [116]. This is not surprising, because a transformation often contains a significant part of the language's semantics. Often, the transformation must be total, *i.e.,* must be able to transform every language concept. Therefore, additive changes will always have to be incorporated in the transformation co-evolution. Levendovszky *et al.* call such additive operations "fully semantic transformation operations", but do not provide support for them.

Garcés *et al.* address image evolution and more extensively domain evolution [68]. Their adaptation chains for migrating transformation models are very similar to our commuting diagrams. Additionally, the projection problem is partly circumvented by using traceability models to keep track of deleted elements. They show how traceability support can be injected into the transformation model by means of a HOT. The approach is implemented on top of Ecore, ATL [92] and AML [67].

Since most of these approaches acknowledge the need for support for manual intervention, Di Ruscio *et al.* present an approach to assess the adaptation cost in case of an evolution [179]. A classification of automated, partially automated and fully semantic adaptations is presented, and dependency links are identified between transformation model and metamodel. This way, the cost can be assessed, and if necessary lowered by refining metamodel changes, similar to our divide-and-conquer strategy.

Other specific co-evolution scenarios that deal with other types of artifacts are studied since our research. Di Ruscio *et al.* tackle the co-evolution of textual concrete syntax specification [178]. Also in this context, they use difference models and transformations to automate co-evolution. The co-evolution of code generators is addressed by Di Rocco *et al.* [174]. Their work is applied to the migration of Acceleo templates that consist mainly of OCL expressions. Automation is achieved by using a HOT. The approach however mainly focuses on refactoring operations.

Recently, Hebig *et al.* advocate a holistic approach as they conducted six case studies in practice to investigate MDE settings, which are similar to our concept of a relational ecosystem but includes tools and artifacts that are not models [83, 82]. Their conclusion is that the composition of a MDE setting is not seldom subject to evolution in practice, and that researches should extend their efforts beyond the metamodel/model or metamodel/transformation relationship. Di Ruscio *et al.* discuss the impact of evolution on different relations in the ecosystem and identify commonalities to define a unifying and comprehensive adaptation process [177].

3.6 Conclusion and Future Work

The lack of support for automated evolution discourages modellers to implement the necessary evolution of languages. In domain-specific modelling especially, modelling languages are used while under development or under on-going change. When such languages evolve, support for (semi-)automated evolution of language artifacts is needed. At the time of this research, previous research had been done on model co-evolution for metamodel evolution. Transformations or semantics had not yet been taken into account. While addressing the problem of modelling language evolution, we made the following contributions in this chapter, which are related to each research question posed in Section 3.1:

— *RQ 2.1.* What is the role of syntax and semantics in language evolution?
We divided evolution into syntactic and semantic evolution. We identified a goal to satisfy consistency and continuity in Section 3.1, effectively meaning that the evolution is syntactically correct (*i.e.,* the conformance relation is preserved throughout the system) and semantically correct (*i.e.,* the system has evolved according to the changes that were intended). We related these concepts to our framework in Section 3.4.3;

— *RQ 2.2.* What is a complete set of language evolution scenarios that incorporates all possible evolution scenarios?
We described four basic evolution scenarios in Section 3.2: model evolution, image evolution, domain evolution and transformation evolution, which can each be handled (semi-)automatically. We showed that the co-evolution of related artifacts can be mapped onto these basic scenarios. The basic evolution scenarios can be combined to address all complex language evolution scenarios;

— *RQ 2.3.* How can transformation models be co-evolved in case of language evolution?
We devised an approach for domain evolution and image evolution in Section 3.2. We identified the projection problem that prevents the full automation of transformation model migration by use of the migration transformation for instance models. We introduced a flexible and modular migration pipeline, that can be reused for instance model migration as well as for transformation model migration. The pipeline offers a high degree of automation by maximally employing coupled migration operations, a high degree of control by isolating manual adaptation, and high execution performance because of the given optimisations. From these building blocks, we reconstructed a solution for transformation evolution in Section 3.3;

— *RQ 2.4.* How can we cover every possible case of language evolution?
We distinguished three levels of automation and we explored all possible consequences of modelling language evolution in a feature diagram in Section 3.4, and discussed the level of automation possible. We provided an algorithm that forms the backbone of our framework. The algorithm uses techniques that are introduced or discussed throughout the chapter and constitutes a coherent solution for the problem of modelling language evolution. The completeness of the framework is achieved by showing how artifacts and their relations can be mapped on a general DSML relational ecosystem. This is backed by the related work done after the time of our research: we could not find any work that cannot be fit into the framework;

— *RQ 2.5.* What are the assumptions and limitations of the approach?
The framework is applicable to DSML relational ecosystems as described in Section 1.2.3. Technological prerequisites for maximally automating the consequences of language evolution are described in Section 3.4.5.

The majority of approaches for language evolution perform best for evolutions that implement refactoring. The other extreme is highly “semantic” evolution, where the meaning of the system changes significantly. Current approaches do not support automation for such evolutions, as the language engineer’s intent is implicit. Therefore, an interesting thread for future work is to convert semantics to syntax, by explicitly modelling it. Chapter 2 addresses this problem partly by providing support for property modelling. A second challenge lies in the relationships between these properties and between (evolutions of) languages, to capture the “intended” changes of a language. Recent work by Barroca *et al.* is an attempt to address this challenge by combining ontology engineering with MPM [12].

CHAPTER 4
Composition of Domain-Specific Modelling Languages

Abstract. Complex systems require descriptions using multiple modelling languages, or languages able to express different concerns, like timing or data dependencies. In this chapter, we propose techniques for the modular definition and composition of languages, including their abstract, concrete syntax and semantics. These techniques are based on (meta-)model templates, where interface elements and requirements for their connection can be established. We illustrate the ideas using the *metaDepth* textual meta-modelling tool. Semantically, language composition can be exceptionally complex. In the domain of heterogeneous model composition, semantic adaptation is the glue that is necessary to assemble heterogeneous models so that the resulting composed model has well-defined and reasonable semantics. We present an execution model for a semantic adaptation between heterogeneous models. We introduce two Domain-Specific Modelling Languages (DSMLs) for explicitly specifying heterogeneous models on the one hand, and rule-based adaptation blocks on the other hand. Both DSMLs are transformed to the *ModHel'X* framework. The DSMLs enables the modeller to easily customise heterogeneous models and semantic adaptations that fit this model in a modular way. We illustrate the use of our DSMLs on a elevator door case study and demonstrate the importance of defining semantic adaptation explicitly by comparing the obtained results to *Ptolemy II*.

4.1 Introduction

Techniques such as metamodelling and model transformation greatly facilitate the development of DSMLs. However, most development methods require the language engineer to start from scratch when developing a new DSML. This is in contrast with the observation that there are similarities between DSMLs: one might need a new variant of Petri nets, a different kind of automaton (as there are tens of variants of these two formalisms), or the combination of a number of formalisms. Therefore, there is a clear need for reuse of existing languages, or language modules as illustrated by Pescador *et al.* [159].

In this chapter we introduce an approach for the modular composition of modelling languages, based on existing language modules. Inspired by generic programming [75, 189], we use existing, generic model composition mechanisms to achieve this [44]. A general terminology to generic programming is given in a comparative study by Garcia *et al.* [69, 70]. In generic programming, abstractions can be formalised in *concepts* as a set of requirements. Concepts can be *refined* by incorporating the requirements concepts, and types can *model* the concept if they adhere to its requirements. In *metaDepth*, these three notions of concepts, refinement and modelling are implemented by the so-called extension mechanisms *concepts*, *extension* and *templates*, respectively [42], which are explained more in detail below. By using these mechanisms, we aim for general applicability of our approach, provided that these composition mechanisms are supported. We support the composition of language modules in the three aspects of a language, abstract syntax, concrete syntax and semantics. Our approach in *metaDepth* supports textual concrete syntax. The following research question summarises this:

RQ 3. Can we extend generic model composition mechanisms, inspired by generic programming, to the composition of DSMLs, in its three aspects, namely, abstract syntax, (textual) concrete syntax and (operational) semantics?

This research question can be further refined into the following research questions:

— *RQ 3.1.* How can abstract syntax be combined using these composition mechanisms?

— *RQ 3.2.* How can textual concrete syntax be combined using these composition mechanisms?

— *RQ 3.3.* How can operational semantics be combined using these composition mechanisms?

— *RQ 3.4.* What are the assumptions and limitations of the approach?

We illustrate the approach with an elevator door example, for which a Timed State Machine language is composed from reusable language modules.

Semantically, composition of languages can be very complex, but necessary in Multi-Paradigm Modelling. A more advanced combination of semantics is necessary as different parts of the system belong to different technical domains, but also because different abstraction levels, different aspects of the system, and different phases in the design require different modelling techniques and tools. It is therefore necessary to be able to cope with multi-paradigm, heterogeneous models, and to give them semantics which are precise enough for simulating the behaviour and validating properties of the system, and to generate as much as possible of the realisation of the system from the model. To illustrate this complexity, we extend the example of an elevator door and show how it can be modelled and simulated using a DSML for heterogeneous modelling. This DSML has a visual concrete syntax and is mapped to *ModHel'X*, a framework for heterogeneous modelling.

A major challenge in model composition is to obtain a meaningful result from models with heterogeneous semantics. Tackling this issue requires that the semantics of the modelling languages used in the composed models is described in a way such that it can be processed and, more importantly, such that it can be composed. In this chapter, we focus on the definition of the "composition laws" for heterogeneous parts of a model, which have to be explicitly described. A key point of our approach is modularity: we should be able to compose heterogeneous models without modifying them. These composition laws are captured in a technique called *semantic adaptation* [23]. In *ModHel'X*, semantic adaptation can be explicitly described separately from models and languages. Often complex semantic adaptations have to be coded however, which is tedious and error prone. We introduce a metamodel for modelling semantic adaptation, founded on an execution model. We then present a DSML that is designed to specify semantic adaptation in an abstract and compact way.

RQ 4. What is a solution according to MPM principles for dealing with execution of heterogeneous modelling and semantic adaptation?

More specifically, we look for an answer to the following research questions:

— *RQ 4.1.* What is an appropriate formalism for heterogeneous modelling?

— *RQ 4.2.* What is an appropriate formalism for semantic adaptation?

— *RQ 4.3.* How is creating semantic adaptation improved by this formalism?

— *RQ 4.4.* What are the assumptions and limitations of the approach?

In this chapter we assume that no overlapping concepts exist between the to be composed language modules. Instead, we assume that the languages are complementary. Hence, we use the term *compose* rather than *merge*. Merging language modules implies resolving overlapping concepts, which can cause conflicts (*i.e.,* constraint violations in the context of modelling) [164]. In this chapter, composition cannot cause conflicts. Rather than focusing on conflict resolution [164, 63] or avoidance [167, 38, 101], the contributions in this chapter are largely focused on the composition of semantics.

This chapter is organised as follows: Section 4.2 addresses the composition of modelling languages with textual concrete syntax and keeps a structural focus. Section 4.3 presents a solution to this in the form of a DSML for semantic adaptation. Subsequently, Section 4.4 discusses related work, and Section 4.5 concludes this chapter and presents directions for future work.

4.2 Composition of Modelling Languages

In this section some (meta)model composition mechanisms of the *metaDepth* tool [44] are examined to see whether they are applicable for language engineering. We use a model of an elevator door as motivating example, and compose the three components of a modelling language: abstract syntax, concrete syntax and operational semantics.

In Section 4.2.1 the mechanisms for composition are introduced. Then, Section 4.2.2 introduces the elevator door example and the Timed State Machine language that will be composed. Subsequently, the abstract syntax is composed in Section 4.2.3, concrete syntax in Section 4.2.4 and semantics in Section 4.2.5. The approach is evaluated in Section 4.2.6, where it is argued that the composition of semantics can only be applied to compositions that are semantically relatively simple.

4.2.1 Composition Mechanisms

In this section we review some (meta)model composition mechanisms [44] that we will use in order to compose modelling languages. These mechanisms are supported by the *metaDepth* tool [42] that we will use throughout this chapter, but the mechanisms can be introduced in other modelling frameworks.

Extension

We use two extension mechanisms, at the class/object level and at the metamodel/model level. Inheritance is used as an extension mechanism for both types (class inheritance) and objects. In the former case, children classes inherit attribute types defined in parent classes. In the latter case, inheritance acts as value overriding. In this way, if children objects do not define a value for an attribute instance, the value is taken from the parent object.

Regarding models, our first modularity mechanism is (meta)model extension, and is applicable to models and metamodels. At the metamodel level, an *extension* is a metamodel fragment that increments the elements in a base metamodel, by adding new types, new constraints, or defining new attributes of existing classes. Therefore, it is similar to *package merge* in UML [53]. This mechanism allows a modular definition of metamodels, and is used intensively, *e.g.,* in the definition of the UML itself [153].

Our extension mechanism works at the model level as well, by using object inheritance. This is useful to define libraries of predefined models, which can be later extended with further objects, or to override the properties of existing objects.

Finally, models and metamodels can use the *import* directive in order to access other model elements.

Concepts

Some of our composition mechanisms are based on expressing *requirements* that need to be fulfilled by other (meta)models, to enable their composition. Using terminology from generic programming [75], we call a *structural concept* a set of structural requirements to be fulfilled by a (meta)model. They can be used to express requirements both for models and metamodels, and therefore have the form of a model or a metamodel as well. For simplicity, this section explains *metamodel concepts*, but explanations are also applicable to *model concepts*. A concept contains classes, attributes and relations that need to exist in a certain metamodel, and therefore the concept elements (classes, attributes, references) are interpreted as variables, which need to be *bound* to the elements of the metamodel.

Concepts are useful in several ways as shown in Figure 4.1. First, operations can be defined over a concept, so that they become reusable. When the concept gets bound to some metamodel, the operation becomes applicable to it. In this case, the concept expresses the requirements for the execution of an operation. Second, concepts can be used to express composition requirements. Hence, it is possible define metamodel fragments, where some elements of such fragments are

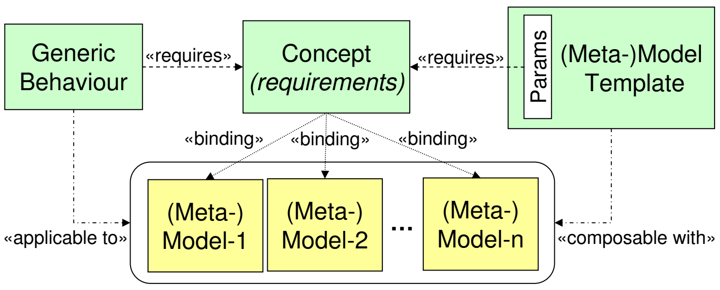


Figure 4.1: Expressing generic behaviour and composability criteria with concepts (from [44]).

variables. The requirements for binding such variables to concrete elements in other models can be expressed through a concept.

A concept *binding* is a process in which the variables in a concept are bound to elements in a metamodel [44]. It forms a one-to-one or many-to-one correspondence between the variables in the concept and elements in the metamodel, allowing some heterogeneity between the concept and the metamodel. For example, the subtyping relations in the concept must be preserved in the metamodel, but the metamodel may collapse several classes, or add intermediate ones.

In order to gain further flexibility, *hybrid concepts* can reduce the structure required from a specific metamodel to a minimum, but instead require the implementation of some operations. Hence, in addition to a binding, the user needs to implement some operations (in EOL). The use of hybrid concepts allows the user to hide unessential structural requirements behind operations, which could, similar to Java interfaces, be implemented in different ways by different metamodels.

In order to create useful concepts, extension is required, because concepts only focus on defining requirements.

Templates

While (meta)model extension provides an extensibility mechanism, such extensions are not a reusability mechanism but a means for modularity. (Meta)model *templates* enable a more flexible definition of model and metamodel fragments, as they permit their connection (perhaps through extension) to other models, which are not specified at design time. Hence, a metamodel template is a metamodel where some elements (metaclasses, features, associations) are variables. The connection requirements for such variables are expressed through a concept.

Operation templates enable defining an operation independently of a concrete metamodel. For this purpose, they are defined over a concept. The template parameters in the operation, which are types, can be used as variables. A binding from the concept to some metamodel assigns a particular type to each variable, and induces a retyping of the operation template, which becomes applicable to the bound metamodel.

In order to create templates, a concept must be defined, which requires extension. It is up to the user to assess whether the most basic composition mechanism *extension* suffices, or the usage of *concepts*, or up the most reusable extension mechanism of using *templates*. It must however be noted that concepts and templates are asymmetric, which means that both artifacts that are to be combined are heterogeneous: one is a regular (meta)model and the other is a concept or template.

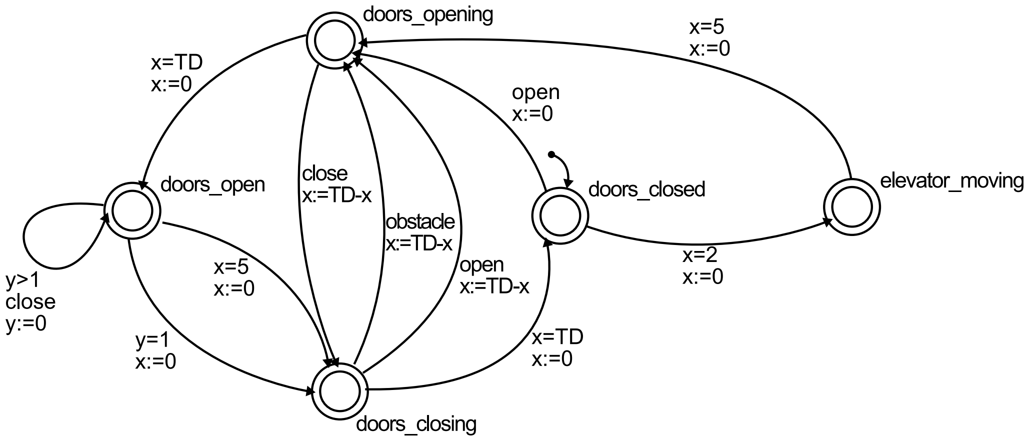


Figure 4.2: A Timed State Machine for an elevator door.

4.2.2 Motivating Example

In this section, we introduce our motivating example based on Timed Automata [3] (see [15] for a more extensive survey). In this chapter, we delve deeper into the behaviour of an elevator door. As a motivating example, we focus on a particular component that is contained in the *Elevator* DSML. We use a model of an elevator door, that manages opening and closing of the door, either prompted by a button press or automatically invoked by the controller itself. This model is a refinement of the door component, and allows us to analyse different properties. In this chapter, we use composition techniques to investigate the most appropriate formalisms to model this component. Since the elevator door model uses time delays and has modal and deterministic behaviour, timed automata appears to be an appropriate formalism to model this component.

A timed automaton is a deterministic, finite-state automaton with a number of clocks. The integer value of each clock increases step-wise, along with the execution of the automaton. A transition may need an input symbol to be enabled. Additionally, a transition of a timed automaton may have a guard expression over the clock values, which disables the transition if it returns to *false*. A transition may also have actions that assign a new value to a clock, after firing the transition. For the sake of simplicity, we assume that the clock value range is unbounded. Theoretically, this automaton decides whether an input word (*i.e.,* a sequence of input symbols) is accepted, if it is in an accepting state after consuming the sequence. In our example, every state is an accepting state so the language recognition possibilities of automata are not used in this chapter. Also, in our example, it is not a requirement that for each state there is an outgoing transition for each symbol. To stress the difference in usage, we call our formalism *Timed State Machines*.

Figure 4.2 shows a model in Timed State Machines of the timed behaviour of an elevator door. The elevator door model can be in one of the following states, for which its outgoing transitions are explained:

— doors_closed: the doors are closed, and the elevator is ready to start moving or the open button can be pressed;

— doors_open: the doors are open, and the only possible action is to close the door, either

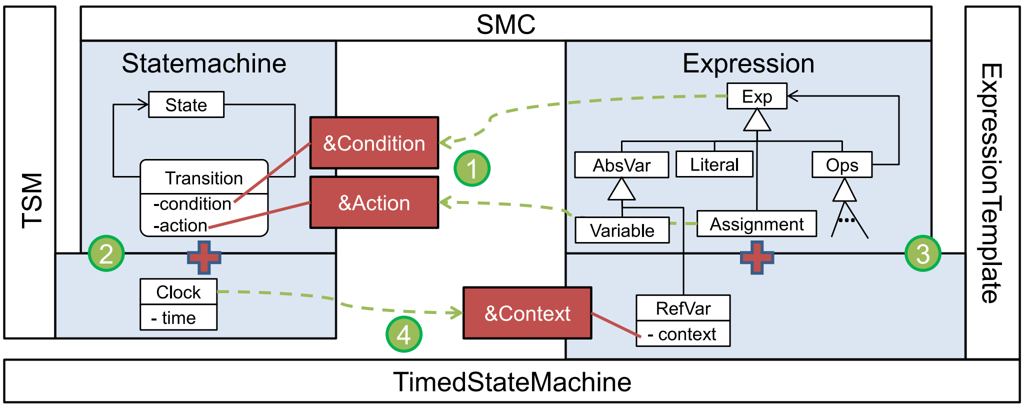


Figure 4.3: The four steps for weaving the two languages.

manually or automatically after a time-out;

— doors_opening: the doors are sliding outward to open, and will eventually be open after a given time delay, or can start closing again if the close button is pressed;

— doors_closing: the doors are sliding inward to close, and will eventually be closed after a given time delay, or can start opening again if the open button is pressed or if an obstacle is detected in the path of the door;

— elevator_moving: the elevator is moving, which means that the doors are blocked. After some given time, the elevator stops and the doors automatically open.

The constant TD represents the time delay for the door to open or close. Two clocks x and y are defined in this automaton. The clock x keeps track of the traversed time in a state and a timed transition can occur when x reaches a certain value (*i.e.*, the controller was a certain time in the current state). To ensure these semantics, x is reset whenever the state changes. The only exceptions are the transitions between doors_opening and doors_closing, because if the doors were already opening for a certain amount of time, they need an equal amount of time to close. For this, we use the formula x:=TD-x. A second clock y is introduced to model a second (parallel) delay in the doors_open state, that occurs after pressing the close button.

The Timed State Machine language is an excellent example of our modular approach for language engineering as it can be seen as a combination of two independent language modules: (a) finite state machines, and (b) a language for expressions, with constructs for comparison, assignment and simple mathematics (for the guards and actions). These two language modules could be used autonomously, or in combination (using our approach) with different language modules. Next, we show how to combine them, and how additional language constructs such as clocks are introduced.

As shown in Figure 4.3, the Timed State Machine language is created from two existing language modules: StateMachine and Expression. Using the composition mechanisms previously presented, we tackle the three aspects of the language: abstract, concrete syntax, and semantics. The two language modules are combined in four steps:

1. SMC: Template instantiation of the existing StateMachine template with the existing Expres-

```
1  Model Expression {
2    Node DeclaredVariable { value : int; }
3    Node Assignment { op1 : AbstractVariable; op2 : Exp; }
4    abstract Node Exp;
5    abstract Node AbstractVariable : Exp;
6    Node Variable : AbstractVariable { context : DeclaredVariable; }
7    Node Literal : Exp { value : int; }
8    Node TrueLit : Exp; Node FalseLit : Exp;
9    abstract Node UnOp : Exp { op1 : Exp; }
10   abstract Node BinOp : Exp { op1 : Exp; op2 : Exp; }
11   Node Equals : BinOp; Node GreaterThan : BinOp; Node SmallerThan : BinOp;
12   Node NEquals : BinOp; Node GEquals : BinOp; Node LEquals : BinOp;
13   Node And : BinOp; Node Or : BinOp; Node Not : UnOp;
14   Node Plus : BinOp; Node Minus : BinOp;
15   Node Multiply : BinOp; Node Divide : BinOp; Node Modulo : BinOp;
16 }
```

Listing 4.1: Abstract syntax model of the Expression module.

sion language, to obtain SMC, a machine with conditions and actions;

2. TSM: Extension of SMC with a Clock language construct, to obtain TSM, a state machine with time;

3. ExpressionTemplate: Extension of TSM to create a new template ExpressionTemplate, linking identifiers in the expressions to a particular value;

4. TimedStateMachine: Template instantiation of ExpressionTemplate to link identifiers with clocks.

(Partial) metamodels are shown in shaded squares representing a language module, with normal arrows as associations and filled arrows as inheritance links. Template parameters are represented by dark squares (with the name of the template variable preceded by "&"), instantiations[1] of template parameters are represented by dotted arrows, and extension is represented by large "+" symbols. Applying these four steps results in TimedStateMachine, for which Associations have conditions and actions that can refer to clock values: *e.g.,* a transition that can only be fired if the value of clock c is greater than 5 (see also Figure 4.2).

4.2.3 Composing Abstract Syntax

In order to create the abstract syntax for the language, the four steps presented above are followed:

A model for the Expression language is shown in Listing 4.1, representing simple algebraic expressions. The language contains a way to declare variables (line 2) and to use them (line 6). An abstract Exp class (line 4) is available which is a superclass of all kinds of expressions (lines 5–15), such as Equals, Plus, Literal (a constant value) or Variable (an identifier). The language also contains an Assignment element (line 3) to assign new values to variables. For simplicity, no distinction is made between Boolean and non-Boolean expressions. This module is a normal *metaDepth* metamodel, but as we will see later, we can still extend it by means of inheritance.

Step 1. Listing 4.2 shows a template of a state machine. It has three template parameters (line 1), which are identifiers preceded by a "&". The first parameter represents a model that contains

[1]Not to be confused with metamodel instantiation.

```
 1 template <&ConditionActionModel, &Condition, &Action>
 2 requires Evaluatable(&ConditionActionModel, &Condition, &Action)
 3 Model StateMachine imports &ConditionActionModel {
 4   Node Symbol;
 5   Node State {
 6     initialflag: boolean = false;
 7     finalflag  : boolean = false;
 8   }
 9   Node Transition {
10     incoming  : State;
11     outgoing  : State;
12     symbol    : Symbol[0..1];
13     condition : &Condition[0..1];
14     action    : &Action[*];
15   }
16   oneInitial  : $State.allInstances().one( s | s.initialflag=true)$
17   severalFinal: $State.allInstances().exists( s | s.finalflag=true)$
18 }
19 StateMachine<Expression, Expression::Exp, Expression::Assignment> SMC;
```

Listing 4.2: Abstract syntax template for the StateMachine module.

```
 1 concept Evaluatable(&Evaluator, &EvaluatableNode1, &EvaluatableNode2) {
 2   Model &Evaluator {
 3     Node &EvaluatableNode1 {
 4       operation eval() : boolean;
 5     }
 6     Node &EvaluatableNode2 {
 7       operation eval();
 8     }
 9   }
10 }
```

Listing 4.3: A hybrid concept to ensure well-formedness of the StateMachines condition and action.

the necessary concepts for conditions and actions and is imported into the StateMachine model (line 3). The second and third parameter represent the condition and action construct, which are then referenced in Transition (lines 13–14): a Transition may have a condition, and multiple actions. A Transition also has an incoming and an outgoing State, and may have a symbol (lines 10–12). A State can be an initial state or a final state (lines 6–7), and there should be exactly one initial state, and at least one final state, as specified by constraints (lines 16–17). Constraints are expressed in EOL-code between dollar signs. The template requires the fulfilment of the Evaluatable concept (line 2). This concept is shown in Listing 4.3 and states that &Condition and &Action should be node elements of a model &ConditionActionModel. Both nodes should implement an operation `eval` (we will come back to this in Section 4.2.5). The `eval` operation of the first node parameter should return a Boolean value, as it represents a condition. The `eval` operation of the first node parameter does not return anything, and will be used as an action.

The template binding is done on line 19 of Listing 4.2. The Exp element of the Expression module can serve as a condition for a Transition, and Assignment can serve as an action for a Transition. All requirements that are imposed by the Evaluatable concept are fulfilled: Exp and Assignment are indeed node elements of Evaluatable, and they do have the correct `eval` operation (see Section 4.2.5). On line 25 of Listing 4.2 the template is instantiated with the Expression

```
1 Model TSM extends SMC {
2   Node Clock { time : int; }
3 }
```

Listing 4.4: Adding the clock feature.

```
 1 concept IntGettable(&Getter, &GettableNode) {
 2   Model &Getter {
 3     Node &GettableNode {
 4       operation getValue() : int;
 5       operation setValue(value : int);
 6     }
 7   }
 8 }
 9 template <&ContextModel, &Context> requires IntGettable(&ContextModel, &Context)
10 Model ExpressionTemplate extends TSM, &ContextModel {
11   Node RefVar : AbstractVariable { context : &Context; }
12 }
13 ExpressionTemplate<TSM, TSM::Clock> TimedStateMachine;
```

Listing 4.5: Linking the Variable element to a clock.

language, with Exp as the condition and Assignment as the action, resulting in SMC, a state machine with conditions and actions.

Step 2. We still need an element denoting a clock, with a clock value. This is achieved by using extension as shown in Listing 4.4. Line 1 states that the defined metamodel TSM extends SMC, the template instantiation shown in line 25 of Listing 4.2. Note that this only adds an element Clock to the metamodel, and no relations with other elements are defined yet.

Step 3. As shown in Listing 4.5, a variable RefVar that can reference variables outside of the Expression module is added via extension. The new RefVar element inherits from the abstract variable class AbstractVariable (abbreviated to AbsVar in Figure 4.3) so that it becomes usable in an expression.

Step 4. The conditions and actions use named variables (*e.g.,* x), which should be able to refer to the clock values in our final timed state machine. This means that RefVar should be linked (*i.e.,* have a context) to a Clock instance. This is shown in Listing 4.5. To this end, the TSM metamodel is extended by means of inheritance (line 11) and extension (line 10), where a RefVar with a context is introduced. A hybrid concept (lines 1–8) defines the requirements for adding a new kind of variable to the language. A template is created (lines 9–12). By using template parameters, the Expression language is once again extended in a generic way. Finally, the template is instantiated with a Clock (line 13).

The language TimedStateMachine that results from the four steps can be instantiated as shown in Listing 4.6, which shows a elevator door *metaDepth* model. All five states (lines 7–11), two of all the transitions (lines 13–16) and both clocks (lines 2–3) from Figure 4.2 can be recognised. There are three input symbols (line 4) and the variable TD is declared (line 5). The conditions and actions are expressions and assignments conforming to the Expression module. They are listed below (line 18–29), and include RefVars to reference clocks (lines 18–19), a Variable to reference the regular variable TD, and literals, assignments and operations. The syntax is quite verbose, especially for the conditions and actions, which become similar to abstract syntax trees. Naturally,

```
TimedStateMachine elevatordoor {
  Clock x { time = 0; }
  Clock y { time = 0; }
  Symbol open; Symbol close; Symbol obstacle;
  DeclaredVariable TD { value = 3; }

  State doors_closed    { initialflag = true; finalflag = true; }
  State doors_open      { finalflag = true; }
  State doors_closing   { finalflag = true; }
  State doors_opening   { finalflag = true; }
  State elevator_moving { finalflag = true; }

  Transition startmove  { incoming = doors_closed; outgoing = elevator_moving;
                          condition = x2; action = [initx]; }
  Transition obs        { incoming = doors_closing; outgoing = doors_opening;
                          symbol = obstacle; action= [TDminx]; }
  //...
  RefVar varx       { context = x; }
  RefVar vary       { context = y; }
  Variable vartd    { context = TD; }
  Literal zero      { value = 0; }
  Literal two       { value = 2; }
  //...
  Assignment initx  { op1 = varx; op2 = zero; }
  Assignment TDminx { op1 = varx; op2 = minus; }
  //...
  Equals x2         { op1 = varx; op2 = two; }
  //...
  Minus minus       { op1 = vartd; op2 = varx; }
}
```

Listing 4.6: The elevator door model as an instance of the TimedStateMachine.

a simpler, more intuitive syntax is desirable, which is defined next.

4.2.4 Composing Concrete Syntax

In *metaDepth*, textual concrete syntax can be defined by creating a TextualSyntax model as described in detail in [43]. Similar to defining the abstract syntax, we use the four steps described above and the extension mechanisms in combination with the TextualSyntax language to obtain the textual syntax model. The TextualSyntax model is compiled to an ANTLR grammar with semantic actions, so that the parsing results in a *metaDepth* model. Just like the abstract syntax model, the concrete syntax model is composed modularly with maximal reuse of existing models. In contrast, the template and composition mechanisms are used at the model level, as we are building and composing instances of the TextualSyntax metamodel. It is important to note however that the compiler to ANTLR does not incorporate inheritance, which can be considered an important tool for the extension mechanism. This limitation will result in some particular design choices in this approach.

The textual concrete syntax model of the Expression module is shown in Listing 4.7. This is a normal textual syntax model as presented in [43], where every model and node have an associated concrete syntax definition. The Expression abstract syntax model of Listing 4.1 is imported (line 2) so that all its elements become accessible. The concrete syntax of the Expression module is defined by assigning the reference model (as a string), name, and a parser rule stating that the Texpr rule should be parsed (line 3). As expected in an expression language, the Texpr rule (line 4) defines its parser rule as a number of declarations, followed by one of all available (by an

```
 1 load "TextualSyntax"
 2 TextualSyntax ExpressionSyntax imports Expression {
 3   TModel TExpression { refModel = "Expression"; name = "Expression"; tempExp = ["&Texpr"]; }
 4   TNode Texpr { tempExp = ["(( Tdecl ))* (( &Tlit | &TabsVar | &Tass | &Teq | &Tgt | ... ))|"]; }
 5   TNode Texpr2 : Texpr {} //  to avoid name clashes if Texpr is used twice in a rule
 6   TNode Tdecl { refNode = DeclaredVariable; tempExp = ["var __id := #value ;"]; }
 7   TNode Tlit { refNode = Literal; tempExp = ["#value"]; }
 8   TNode Tvar { refNode = Variable; tempExp = ["< #context >"]; }
 9   TNode Tass { refNode = Assignment;
10     tempExp = ["&TabsVar := &Texpr"]; creationExp = ["#op1 = &TabsVar", "#op2 = &Texpr"]; }
11   TNode Teq { refNode = Equals;
12     tempExp = ["( &Texpr = &Texpr2 )"]; creationExp = ["#op1 = &Texpr", "#op2 = &Texpr2"]; }
13   TNode Tgt { refNode = GreaterThan;
14     tempExp = ["( &Texpr > &Texpr2 )"]; creationExp = ["#op1 = &Texpr", "#op2 = &Texpr2"]; }
15   // ...
16   genPackage = "serializers";
17   fileExtension = "expression";
18   permitsLoad = true;
19 }
```

Listing 4.7: TextualSyntax model that describes concrete syntax for Expression.

or-operation, abbreviated) possible rules. Texpr2 is defined inheriting the parser rule of Texpr (line 5), in order to allow two uses of Texpr in subsequent parser rules. This is a technical issue: if twice the same reference is used, a name clash will occur in the generated ANTLR code. Below, all subrules are defined as recursive rules referring the Texpr rule. The TNode instances consist of the node they will generate when this rule is parsed, an expression tempExp denoting the parser rule that can directly reference attributes or use references to other parser rules, and an optional number of expressions creationExp that assign results of referenced parser rules to attributes. In the expression language, brackets are mandatory because the model is converted to a left-recursive parser. Finally, some parser options are specified (lines 16–18).

It is important to note that TabsVar is not defined at this time. TabsVar will serve as an entry point for later extension and is therefore currently left undefined. The reason for this was briefly mentioned before: in TextualSyntax models, unfortunately, rules that inherit from an abstract TNode such as TabsVar are not taken into account when TabsVar is used as reference in a tempExp. If one wants to use the Expression module autonomously, one would have to extend this textual syntax model with:

```
TNode TabsVar { tempExp = ["&Tvar"]; }
```

Step 1. Listing 4.8 shows a concept and template for the concrete syntax of the StateMachine module of Listing 4.2. It defines the syntax for a StateMachine module (lines 9–10), Symbol instances (lines 13–14), State instances (lines 15–19) and Transitions (lines 21–26). The syntax for states and transition each consists of several possibilities, each modelled as its own rule. Analogue to the StateMachine module, its concrete syntax consists of a concept and a model template, so that the parser rules for Transition can reference the parser rule defined for the template parameters &TCond and &TAct. They will become available when the model template is instantiated (line 29), and additional rules are defined to make them usable in parser rules (lines 11–12), because template parameters cannot be directly used in parser rules. The template is instantiated using ExpressionSyntax (see Listing 4.7). The textual syntax of the conditions and actions are the template parameters &TCond and &TAct, that have to be TNodes in TextualSyntax model &TConditionModel according to the ModelTNode2 concept (lines 1–6).

```
concept ModelTNode2(&ModelSyntax2, &theTNode1, &theTNode2) {
  TextualSyntax &ModelSyntax2 {
    TNode &theTNode1;
    TNode &theTNode2;
  }
}
template <&TConditionModel, &TCond, &TAct> requires ModelTNode2(&TConditionModel, &TCond, &TAct)
TextualSyntax StateMachineSyntax imports StateMachine, &TConditionModel {
  TModel TMachine { refModel = "StateMachine"; name = "StateMachine";
    tempExp = ["StateMachine __id { (( (( &TSymbol | &TState | &TTransition ))| ))* } "]; }
  TNode Tcond : &TCondition;
  TNode Tact : &TAction;
  TNode TSymbols { tempExp = ["alphabet (( &TSymbol ))+ ;"]; }
  TNode TSymbol { refNode = Symbol; tempExp = ["__id"]; }
  TNode TStates { tempExp = ["states (( (( &TState1 | &TState2 | &TState3 | &TState4 ))| ))+ ;"]; }
  TNode TState1 { refNode = State; tempExp = ["__id"]; }
  TNode TState2 { refNode = State; tempExp = ["> __id"]; creationExp = ["#initialflag = true"]; }
  TNode TState3 { refNode = State; tempExp = ["__id !"]; creationExp = ["#finalflag = true"]; }
  TNode TState4 { refNode = State;
    tempExp = ["> __id !"]; creationExp = ["#initialflag = true", "#finalflag = true"]; }
  TNode TTransition {
    tempExp = ["(( &TTransition1 | &TTransition2 | &TTransition3 | &TTransition4 ))|"]; }
  // ...
  TNode TTransition3 { refNode = Transition;
    tempExp = ["#incoming -> #outgoing #symbol [ &Tcond ] / (( [ &Tact ] ))* ;"];
    creationExp = ["#condition = &Tcond", "#action = &Tact"]; }
  // ...
}
StateMachineSyntax<ExpressionSyntax, ExpressionSyntax::Texpr, ExpressionSyntax::Tass> SMCSyntax;
```

Listing 4.8: TextualSyntax model that describes concrete syntax for StateMachine.

```
TextualSyntax TSMSyntax imports TSM, SMCSyntax {
  TNode TClock {
    refNode     = Clock;
    tempExp     = ["clock __id #time ;"];
  }
}
```

Listing 4.9: Adding the clock feature.

Step 2. Listing 4.9 shows how extension is used to define a concrete syntax for a Clock, resulting in the concrete syntax of the TSM module.

Step 3. As shown in Listing 4.10, a parser rule for the reference variable RefVar is added to the concrete syntax of the ExpressionTemplate module (line 12). Since inheritance is not supported in parser rules, TabsVar is defined in this template as either applying the Tvar parser rule of the original Expression concrete syntax model, or as the new TrefVar rule (line 11). Note how the TModel parser rule replaces the one of the StateMachine concrete syntax model of Listing 4.8, and adds the newly defined element references. Since parser rules are represented as strings, this is the only way to extend the model parser. Nonetheless, the modularity of the parser expressions keep the impact of this replacement limited.

Step 4. The parser rule for Clock is linked to the concrete syntax model by defining a concept stating that a concrete syntax model with a TNode should be provided (lines 1–5). On line 16, the above described template is instantiated with the TClock parser rule resulting in a concrete syntax model for TimedStateMachine. On lines 12–14 it is stated where to save the generated code, that the file extension for a TimedStateMachine is “.tsm” and that TimedStateMachine can be loaded in

```
concept ModelTNode(&ModelSyntax, &theTNode) {
  TextualSyntax &ModelSyntax {
    TNode &theTNode;
  }
}
template <&TContextModel, &TContext> requires ModelTNode(&TContextModel, &TContext)
TextualSyntax ExpressionTemplateSyntax imports TimedStateMachine, TSM, &TContextModel {
  TModel TTimedStateMachine { refModel = "TimedStateMachine"; name = "TimedStateMachine";
    tempExp = ["TimedStateMachine __id { (( (( &Tdecl | &TSymbols | &TClock | &TStates | &TTransition ))| ))* } "]; }
  TNode TabsVar { tempExp = ["(( &TrefVar | &Tvar ))|"]; }
  TNode TrefVar { refNode = RefVar; tempExp = ["#context"]; }
  genPackage = "serializers";
  fileExtension = "tsm";
  permitsLoad = true;
}
ExpressionTemplateSyntax<TSMSyntax, TSMSyntax::TClock> TimedStateMachineSyntax;
```

Listing 4.10: Linking the Variable element to a clock.

```
 1 TimedStateMachine elevatordoor {
 2   alphabet open close obstacle;
 3   clock x 0;
 4   clock y 0;
 5   var TD := 3;
 6   states doors_open! doors_closing! >doors_closed! doors_opening! elevator_moving!;
 7   doors_open->doors_open close [(y>1)] / [y:=0];
 8   doors_open->doors_closing [(x=5)] / [x:=0];
 9   doors_open->doors_closing [(y=1)] / [x:=0];
10   doors_closing->doors_opening open / [x:=(<TD>-x)];
11   doors_closing->doors_opening obstacle / [x:=(<TD>-x)];
12   doors_closing->doors_closed [(x=<TD>)] / [x:=0];
13   doors_closed->elevator_moving [(x=2)] / [x:=0];
14   doors_closed->doors_opening open / [x:=0];
15   elevator_moving->doors_opening [(x=5)] / [x:=0];
16   doors_opening->doors_open [(x=<TD>)] / [x:=0];
17   doors_opening->doors_closing close / [x:=(<TD>-x)];
18 }
```

Listing 4.11: The elevator door model using the modularly composed concrete syntax.

metaDepth.

Listing Listing 4.11 shows the elevator door model of Figure 4.2 as an instance of the TimedStateMachine, using the concrete syntax. The model is much more concise and clear, compared to the same model using the default abstract syntax description (see Listing 4.6), especially for the conditions and actions. Due to the limits of the TextualSyntax language and the associated left-recursive parser, some inconveniences remain, such as using brackets around every (sub)expression, using angle brackets to reference declared variables (line 11), and using semicolons to end notations. In this approach, no support is provided for parsing error notification, syntax highlighting, automatic completion or other typical parser extensions for textual syntax. In order to implement such support, the underlying ANTLR data structures can be used. Nonetheless, according to MPM principles, support should be provided in metaDepth, at the modelling level.

4.2.5 Composing Operational Semantics

Abstract and concrete syntax are composed following the same outline. This is possible, as they are both implemented as *metaDepth* models (metamodels in the case of the abstract syntax).

```
 1 operation DeclaredVariable     setValue(value : Boolean)          { self.value := value; }
 2 operation DeclaredVariable     getValue()                 : Boolean { return self.value; }
 3 operation Literal     eval() : Integer { return self.value; }
 4 operation TrueLit     eval() : Boolean { return true; }
 5 operation FalseLit    eval() : Boolean { return false; }
 6 operation Assignment  eval()           { self.op1.context.setValue(self.op2.eval()); }
 7 operation Equals      eval() : Boolean { return self.op1.eval() = self.op2.eval(); }
 8 // ...
 9 operation Minus       eval() : Integer { return self.op1.eval() - self.op2.eval(); }
10 // ...
11 operation Variable    eval() : Integer { return self.context.getValue(); }
```

Listing 4.12: Operations for the Assignment and Equals elements of Expression.

```
concept ExecutableTimedStateMachine(&M, &S, &T, &C) {
  Model &M {
    Node &S {
      operation isInitial() : boolean;
      operation isFinal() : boolean;
    }
    Node &T {
      operation incoming() : &S;
      operation outgoing() : &S;
      operation canFire(symbol : int) : boolean;
      operation action();
    }
    Node &C {
      operation getValue() : int;
      operation setValue(value : int);
    }
  }
}
```

Listing 4.13: Concept for binding semantics to a Timed State Machine.

This means that the same composition mechanisms can be used. In this chapter, semantics are implemented as EOL programs, a generic way of implementing operational semantics. This means that the approach is not limited (nor tailored) to operational semantics as *e.g.,* rule-based transformations. Since semantics are implemented as EOL programs, other mechanisms, namely hybrid concepts, operation templates and binding, have to be used.

The operational semantics of the expression module is shown in Listing 4.12. Operations are eval methods that can be called on all language constructs (lines 3–11). For every operands, eval operations are recursively called. For DeclaredVariable instances, getValue and setValue operations can be called returning or setting the value (lines 1–2). These are used when evaluating Assignments (line 6) and Variables (line 6).

Step 1. A concept ExecutableTimedStateMachine is defined in Listing 4.13, representing requirements for a Timed State Machine simulator. This is a hybrid concept, as it requires both a binding and an implementation of some operations. The concept requires a Model M and Nodes &S modelling states, &T modelling transitions, and &C playing the role of clocks. For each element, the concept defines which operations need to be implemented as they will be used by the simulator. This way, semantics are decoupled from any specific metamodel. To make the associated simulator executable, a binding is required, plus the implementation of some operations. In contrast to the abstract syntax and concrete syntax after step 1, the clocks already have to be incorporated in the

```
 1 @concept(name="ExecutableTimedStateMachine")
 2 operation main() {
 3   // ...
 4   current := &S.select( s | s.isInitial() ).first();
 5   ws := 0;
 6   while (ws < input.size()) {
 7     ('step='+ws.toString()+' current='+current.name+' ').print();
 8     symb := input.at(ws);
 9     ws := ws+1;
10     for (c in &C.all()) { (c + '=' + c.getValue()+' ').print(); }
11     ('symbol=<'+symb+'> ').print();
12     x := &T.all().select( t | t.incoming()=current and t.canFire(symb) );
13     if (x.size()>0) {
14       'TRANSITION'.println();
15       currenttrans := x.first();
16       current := currenttrans.outgoing();
17       for (a in currenttrans.action) a.eval();
18     } else {
19       ('SKIP').println();
20     }
21     for (c in &C.all()) c.setValue(c.getValue()+1);
22   }
23 }
```

Listing 4.14: Simulator for Timed State Machines.

```
bind Evaluatable(SMC, SMC::Exp, SMC::Assignment) requires "ExpressionOps.eol"
```

Listing 4.15: Binding the operations to the Evaluatable language.

template according to this concept. The coupling can be chosen to be more loose however, such as having a more generic doStep method, rather than a more limiting getValue and setValue.

Listing 4.14 shows the core of the semantics for the state machine language. The operation uses the template parameters defined in the previous concept, and their operations, as stated in line 1. First, the initial state is set as the currently active state (line 4). Each iteration of the while-loop (lines 6–22), the next symbol on a given input sequence is fed to the state machine. This is done by first finding all enabled transitions (line 12). If there is at least one enabled transition (although not specifically handled here, more than one transition should not be enabled as this language is nondeterministic), the first (only) one is fired (line 16), and all associated actions are triggered (line 17). In any case, even if there is no enabled transition, all clock values are incremented (line 21). When all input symbols are consumed, the simulator stops.

Listing 4.15 shows how the operations of Listing 4.12, implemented in the file *ExpressionOps.eol*, are bound to the SMC module. The Evaluatable concept was already shown in Listing 4.3. As a result, the eval operations become available to the simulator.

These eval operations are used in the implementation of the operations declared in the concept shown in Listing 4.13, such as isInitial, canFire or setValue. Listing 4.16 shows implementations of the operations for the TimedStateMachine metamodel. The operation canFire (lines 5–7) and action (line 8) in particular make use of the eval operations. The first four operations (lines 1–4) are helper operations to enhance the reusability of the simulator. The simulator also makes use of eval operations on line 17.

Step 2. Listing 4.17 shows implementations of the operations on Clocks. The time attribute

```
operation State isInitial() : Boolean { return self.initialflag; }
operation State isFinal() : Boolean { return self.finalflag; }
operation Transition incoming() : State { return self.incoming; }
operation Transition outgoing() : State { return self.outgoing; }
operation Transition canFire(symbol : Symbol) : Boolean  {
  return (not self.symbol.isDefined() or self.symbol = symbol)
       and (not self.condition.isDefined() or self.condition.eval()); }
operation Transition action() { for (action in self.action) { action.eval(); } }
```

Listing 4.16: Operations needed for state machine semantics.

```
operation Clock getValue() : Integer { return self.time; }
operation Clock setValue(value : Integer) { self.time := value; }
```

Listing 4.17: Operations needed for the Clock construct.

```
operation RefVar eval() : Integer { return self.context.getValue(); }
```

Listing 4.18: Operation needed to get a clock value.

```
bind ExecutableTimedStateMachine(TimedStateMachine, TimedStateMachine::State,
                                 TimedStateMachine::Transition, TimedStateMachine::Clock)
requires "TimedStateMachineOps.eol"
```

Listing 4.19: Binding the operations to the ExecutableTimedStateMachine language.

can be set or retrieved.

Step 3. Listing 4.18 shows an implementation of the added eval operation For the RefVar element. As a result, RefVars can now be evaluated and their time can be set or retrieved. This means that the eval operation of the Assignment element is also applicable, as it sets a context value through the setValue operation.

Step 4. Finally, Listing 4.19 shows the binding of the concept to the TimedStateMachine metamodel defined in Listing 4.5. All parameters need to structurally correspond to the concept, an must have the required operations. Note that both conditions and actions (which is depicted as step 1 in Figure 4.3), and the clock are bound in this step.

4.2.6 Evaluation

We have demonstrated the use of different composition mechanisms to define abstract syntax, concrete syntax and semantics of a language in a modular way. Abstract syntax and concrete syntax are defined through metamodels and models, and therefore template models and concepts are used. Semantics are defined differently, through an EOL program. They can be defined using a hybrid concept, which gathers the elements and operations needed by the simulator template.

Listing 4.20 shows a trace of executing the model of Figure 4.2, after parsing the model of Listing 4.11. A line is printed after every execution step (*i.e.,* an iteration of the while loop in the simulator), and represents a second. A certain input sequence of symbols is given, which is consumed by the model in a step-by-step manner. The current symbol (often none) at every

```
step=0 current=doors_closed y=0 x=0 symbol=<> SKIP
step=1 current=doors_closed y=1 x=1 symbol=<> SKIP
step=2 current=doors_closed y=2 x=2 symbol=<> TRANSITION
step=3 current=elevator_moving y=3 x=1 symbol=<> SKIP
step=4 current=elevator_moving y=4 x=2 symbol=<> SKIP
step=5 current=elevator_moving y=5 x=3 symbol=<> SKIP
step=6 current=elevator_moving y=6 x=4 symbol=<> SKIP
step=7 current=elevator_moving y=7 x=5 symbol=<> TRANSITION
step=8 current=doors_opening y=8 x=1 symbol=<close> TRANSITION
step=9 current=doors_closing y=9 x=3 symbol=<> TRANSITION
step=10 current=doors_closed y=10 x=1 symbol=<> SKIP
step=11 current=doors_closed y=11 x=2 symbol=<> TRANSITION
step=12 current=elevator_moving y=12 x=1 symbol=<> SKIP
step=13 current=elevator_moving y=13 x=2 symbol=<> SKIP
step=14 current=elevator_moving y=14 x=3 symbol=<> SKIP
step=15 current=elevator_moving y=15 x=4 symbol=<> SKIP
step=16 current=elevator_moving y=16 x=5 symbol=<> TRANSITION
step=17 current=doors_opening y=17 x=1 symbol=<> SKIP
step=18 current=doors_opening y=18 x=2 symbol=<close> TRANSITION
step=19 current=doors_closing y=19 x=2 symbol=<> SKIP
step=20 current=doors_closing y=20 x=3 symbol=<> TRANSITION
step=21 current=doors_closed y=21 x=1 symbol=<open> TRANSITION
step=22 current=doors_opening y=22 x=1 symbol=<> SKIP
step=23 current=doors_opening y=23 x=2 symbol=<> SKIP
step=24 current=doors_opening y=24 x=3 symbol=<> TRANSITION
step=25 current=doors_open y=25 x=1 symbol=<> SKIP
step=26 current=doors_open y=26 x=2 symbol=<> SKIP
step=27 current=doors_open y=27 x=3 symbol=<> SKIP
step=28 current=doors_open y=28 x=4 symbol=<> SKIP
step=29 current=doors_open y=29 x=5 symbol=<> TRANSITION
step=30 current=doors_closing y=30 x=1 symbol=<> SKIP
step=31 current=doors_closing y=31 x=2 symbol=<obstacle> TRANSITION
step=32 current=doors_opening y=32 x=2 symbol=<> SKIP
step=33 current=doors_opening y=33 x=3 symbol=<> TRANSITION
step=34 current=doors_open y=34 x=1 symbol=<> SKIP
step=35 current=doors_open y=35 x=2 symbol=<> SKIP
step=36 current=doors_open y=36 x=3 symbol=<close> TRANSITION
step=37 current=doors_open y=1 x=4 symbol=<> TRANSITION
step=38 current=doors_closing y=2 x=1 symbol=<> SKIP
step=39 current=doors_closing y=3 x=2 symbol=<> SKIP
step=40 current=doors_closing y=4 x=3 symbol=<> TRANSITION
step=41 current=doors_closed y=5 x=1 symbol=<> SKIP
step=42 current=doors_closed y=6 x=2 symbol=<> TRANSITION
step=43 current=elevator_moving y=7 x=1 symbol=<> SKIP
step=44 current=elevator_moving y=8 x=2 symbol=<> SKIP
step=45 current=elevator_moving y=9 x=3 symbol=<open> SKIP
step=46 current=elevator_moving y=10 x=4 symbol=<> SKIP
step=47 current=elevator_moving y=11 x=5 symbol=<> TRANSITION
step=48 current=doors_opening y=12 x=1 symbol=<> SKIP
step=49 current=doors_opening y=13 x=2 symbol=<> SKIP
step=50 current=doors_opening y=14 x=3 symbol=<> TRANSITION
step=51 current=doors_open y=15 x=1 symbol=<> SKIP
step=52 current=doors_open y=16 x=2 symbol=<> SKIP
step=53 current=doors_open y=17 x=3 symbol=<open> SKIP
step=54 current=doors_open y=18 x=4 symbol=<> SKIP
```

Listing 4.20: An execution trace of the elevator door model of Figure 4.2.

step is shown between angle brackets after `symbol`. Furthermore, the current state is given, the clock values are given and it is shown whether or not a transition is fired. The trace shows that the elevator autonomously starts moving, and after five seconds it stops moving and starts opening the door. After one second, the close button is pressed, causing the door to close for one second. Further in the trace, a closing door is opened again with a open button (step 21) and the detection of an obstacle (step 31). Also, it is shown how manually closing the door happens after a delay (step 36–38). Furthermore, it is shown that pressing the open button does not have effect when

	Step 1 (SMC)	*Step 2 (TSM)*	*Step 3 (Expression-Template)*	*Step 4 (TimedState-Machine)*
Abstract syntax	hybrid concept + template	extension	extension	hybrid concept + template
Concrete syntax	concept + template∗	extension	extension	concept + template
Semantics	operation template + binding†	operations	operations	hybrid concept + binding†

Table 4.1: Comparison of the degree of automation between the state of the art and *ProMoBox*.

moving (line 45) and when the doors are already open (line 53).

Table 4.1 summarises the use of composition mechanisms for each step. The correspondence between abstract syntax and concrete syntax is clear. For abstract syntax, a hybrid concept is used rather than a regular concept, to incorporate requirements about semantics, but apart from this, the same mechanisms are used. One remark ∗ has to be made about defining the concrete syntax: a parser rule for AbsVar must be intentionally left out to support later extension of the concrete syntax model. Nonetheless, it can be concluded that, using the composition mechanisms, abstract syntax and concrete syntax both can be combined in a modular, structured way.

The correspondence with the semantics is less straightforward however. The remarks † in the Table 4.1 represent the issue that the actual binding of step 1 is only done in step 4, thus entangling the different steps, at the expense of modularity. Also, the semantics of the clocks have to be introduced in the simulation loop in step 1. This illustrates the core of the problem: a more complex entanglement of semantics cannot be easily supported when using a programming language. Many solutions exist in programming to tackle this problem, such as aspect-oriented programming [97]. In the next section we investigate, according to MPM principles, an approach that is specific to the simulation of modelling languages.

Additionally, the trace of Listing 4.20 reveals that the execution is not optimal. The Timed State Machine language combines event-based logic (*i.e.,* pressing buttons an timed events) with continuous-time concepts (*i.e.,* opening and closing the door). To obtain a less coarse execution of the system, an execution step could be shortened to one millisecond or microsecond, but this exposes the weakness of the approach: every step would need an entire evaluation of the model, even if nothing is happening at that step. Clearly, the combination of heterogeneous formalisms is not addressed with this approach.

In the next section, a solution to the combination of semantics of (heterogeneous) languages is presented. Because of the inherent heterogeneity of the to be combined formalisms, the approach will not combine simulators at the metalevel as done in this section, but will rather present a way to communicate between heterogeneous simulators.

4.3 A DSML and Execution Semantics for Semantic Adaptation

As concluded in the previous section, there is a need for composing models with heterogeneous semantics, especially in the context of multi-paradigm modelling. We aim for a modular approach, meaning that we should be able to compose heterogeneous models without modifying them. To illustrate semantic adaptation, the elevator door example is adapted. It is subsequently modelled and simulated using existing tools for heterogeneous modelling. The example is used to present a DSML for heterogeneous modelling and another DSML for semantic adaptation.

The principles of heterogeneous modelling and semantic adaptation are explained in Section 4.3.1. A more realistic version of the running example is presented in Section 4.3.2. Next, a DSML for heterogeneous modelling, based on *ModHel'X*, is introduced in Section 4.3.3. In Section 4.3.4 a second DSML is presented for the textual modelling of semantic adaptation, which is put into operation in Section 4.3.5. A final discussion on semantic adaptation is presented in Section 4.3.6.

4.3.1 Semantic Adaptation

The composition of semantics in the context of simulation is called *semantic adaptation*. in this technique, composition laws are defined to enable run-time communication of heterogeneous simulators, represented as *models of computation* in this context.

In [23, 24], the authors propose to consider semantic adaptation between heterogeneous parts of a model along three axes:

- **Adaptation of data.** The adaptation of data consists in transforming data from a modelling paradigm to data in another paradigm at the boundary between heterogeneous parts of a model. For instance, an event with a value may be transformed into an input symbol for a state machine according to some decoding table;
- **Adaptation of control.** The adaptation of control describes when different parts of the model should be *given control*, so that they can process their inputs and produce outputs. This can be quite complex because the requirements of different models of computation for the execution of a model can interfere in complex ways. For instance, a model that processes a periodically sampled signal should be given control only at its sampling instants, not when some input becomes available for it. Some components may require that all of their inputs be available before they are given control. For other components, the relationship between the availability of inputs and control may be conditional. For instance, a logical OR gate needs to know the value of its second input only when the other input is *false*. Control may also depend on a mix of the occurrence of an event and meeting a deadline, for instance when a state machine has both regular and timed transitions leaving its current state;
- **Adaptation of time.** The adaptation of time takes care of modelling the different time scales used in different parts of an heterogeneous model. We consider here that time may be multiform [5] and counted in seconds in one model, in degrees of rotation in another, and in meters in a third one. The semantic adaptation of time can be used to model the relationship between the crankshaft and camshaft rotation angles in an engine, or to model the different paces at

which time flows in GPS satellites and in a GPS receiver on ground as predicted by general relativity.

The design of reusable semantic adaptation modules is difficult and error prone. Semantic adaptation aims at defining a framework to reduce the semantic gap between adaptation behaviours and low level coding in a general purpose language. The decomposition along three axes and the use of models of time from MARTE [4] and CCSL [71] were a first step presented by Boulanger *et al.* [23]. This work takes this a step further with an execution model for semantic adaptation, all supported by DSMLs.

ModHel'X is a framework for modelling and simulating heterogeneous models developed by Boulanger and Hardebolle *et al.* [23, 81]. *ModHel'X* allows for modular semantic adaptation by explicitly defining an *interface block* between the parent model and the embedded model. In this framework, semantic adaptation is modelled explicitly in a modular way using interface blocks [23], and is considered along the three axes of data, control and time. The adaptation of time is necessary when two models use different time scales. Interface blocks are similar to tag conversion actors as presented in [28], but can perform more complex operations. In *ModHel'X*, control and time adaptation rely on a model of time which is inspired from the MARTE UML profile [4, 71], and can be considered as a restriction of the Clock Constraints Specification Language [71], extended with time tags [22]. In this chapter, we use the name *adaptation block* instead of interface block, to stress that the adaptation block is not just an interface, but also contains behaviour.

4.3.2 Motivating Example Revisited

As discussed in Section 4.2.6, the elevator door example semantically consists of two different models: a controller representing the reactive behaviour, which actuates a plant model representing a physical process. Therefore, modelling the elevator door in the Timed State Machines formalism is not in accordance of MPM's principle of modelling in the most appropriate formalism. A more accurate solution is to model an elevator door controller as a Timed Finite State Machine (TFSM) and the plant model as a Synchronous Data Flow (SDF) model.

The TFSM model differs from the Timed State Machines formalism of Section 4.2 in that there is no concept of variables, clocks and expressions. A TFSM reacts purely on events, and can as an action send events. Additionally, a transition can fire after a given timeout. All control logic of the elevator door is implemented in this formalism. It models how the system reacts to button presses or predetermined timeouts. As a result, the TFSM formalism is more appropriate for modelling reactive behaviour with time, without the additional calculus of clock timeouts as with the Timed State Machines formalism. Only "pure" timeouts are used here: in the example only one timeout is defined, which models that the doors automatically close after 5 seconds. For simplicity, the delay after pressing the close button when the elevator doors are open is left out in this example. Timeouts that are inferred from other events, such as opening or closing doors, are not part of the elevator door controller. Instead, events can be sent to the controller indicating that the door is open or closed, or that the elevator is moving or has stopped. Together with the tree input events of Figure 4.2 (open, close and obstacle), this totals in seven possible input events. The output events are related to the motor of the elevator door: it can go inward to close (motor_in), outward to open (motor_out) or stop (motor_stop).

The dynamics of the plant model for opening and closing an elevator door could be modelled

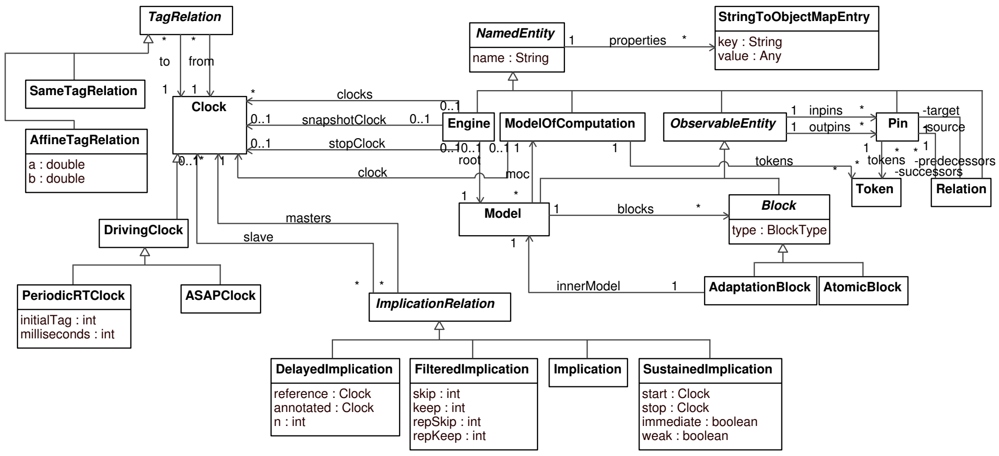


Figure 4.4: The *ModHel'X* metamodel (adapted from [23]).

using differential equations, but for making the example easier to understand, we discretise its behaviour and represent it using difference equations, for which a SDF is suitable. SDF is a data flow formalism in which data represents discretised continuous signals. Data flows over arcs through blocks, which represent functions. Continuity in SDF is approximated by using a fixed time step. In our example, the plant model calculates according to an incoming motor signal, the position of the door at a given time and whether the door is completely open or completely shut. For simplicity, no acceleration is taken into account in this model.

Next, an implementation in *AToM*^{3} for heterogeneous modelling is presented in which the above described example is implemented. The remaining part of this section will focus on how these to models are simulated together by means of adaptation blocks.

4.3.3 A DSML for Heterogeneous Modelling

Although models and adaptation blocks can be programmed with *ModHel'X*, the language engineer still has to code his/her models in Java, which is perceived tedious and error prone. Simple modelled systems easily take several hundreds of lines of code. To this end, we introduce a DSML for modelling block diagrams in *AToM*^{3}. *AToM*^{3} is the most appropriate tool for this task, as an advanced concrete syntax (*i.e.,* geometric-based rather than connection-based [37]) is defined.

DSML Metamodel

The metamodel for the DSML for heterogeneous modelling is shown in Figure 4.4 and is heavily influenced by the design of *ModHel'X*. Unsurprisingly, the semantics of the DSML is a mapping to *ModHel'X*. This mapping is relatively straightforward, as most DSML elements directly correspond to *ModHel'X* counterparts. Since *ModHel'X* revolves around generic modelling, a Model has an associated ModelOfComputation that contains its execution algorithm. A Model also consists of a number of Blocks that have a given BlockType. Blocks can be AtomicBlocks, that represent a simple function, or AdaptationBlocks that contain another Model. Semantic adaptation is implemented

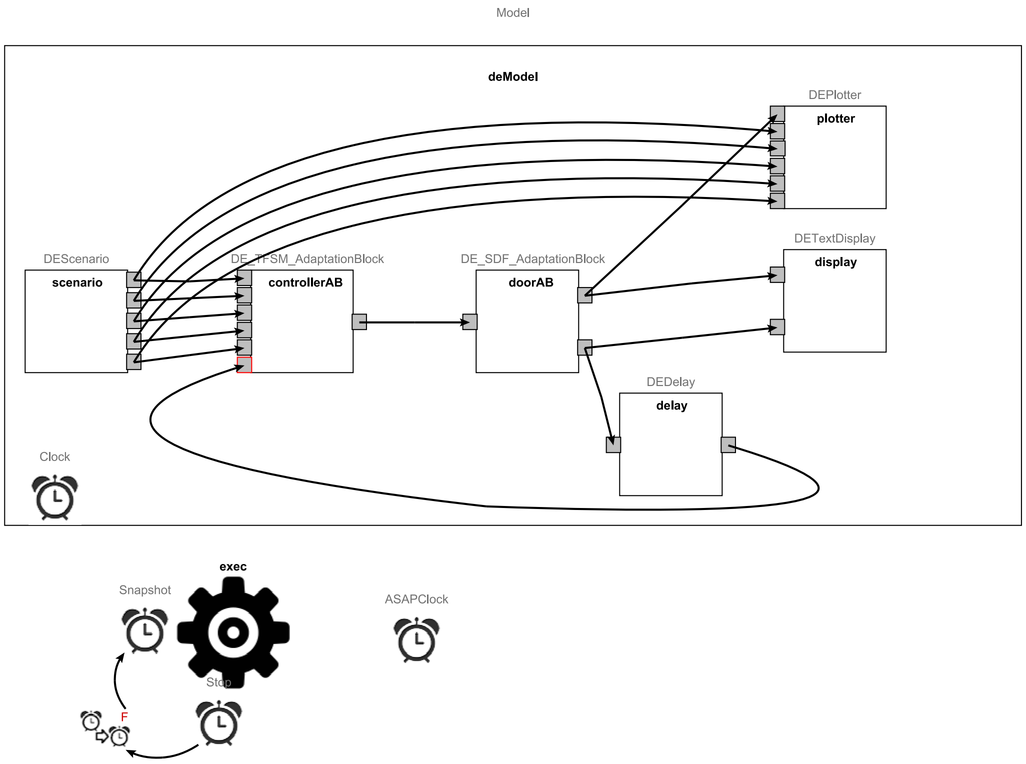


Figure 4.5: The main DE model of an elevator door in *AToM³*.

in AdaptationBlocks, which thus introduces a hierarchical composition of models. Models and Blocks alike can be connected through their input and output Pins. Output Pins of one Block or Model can be connected to input Pins of another. Pins can have data Tokens on them during simulation, which are managed by the ModelOfComputations. All NamedEntitys have properties to represent static parameters (as name-value pairs), which are different depending on the BlockType or ModelOfComputation. An Engine starts the simulation of the root Model, according to its ModelOfComputation. This ModelOfComputation manages data (how data values are passed), control (when which block is simulated) and time (at what moment simulation steps need to be taken) between all Blocks in the Model. If a Block is an AdaptationBlock, it adapts data, control and time for its embedded innerModel. The Engine and the ModelOfComputations rely on Clocks that are of a specific type to keep track of advanced time during simulation. Clocks can have several types of TagRelations (describing a relationship between clocks with a tag calculus to define different clock scales) and ImplicationRelations (describing causal relationships between a master and a slave Clock). The specific Clocks and relations that are used in the example are explained below. A more detailed explanation on this metamodel and the workings of *ModHel'X* can be found in [23].

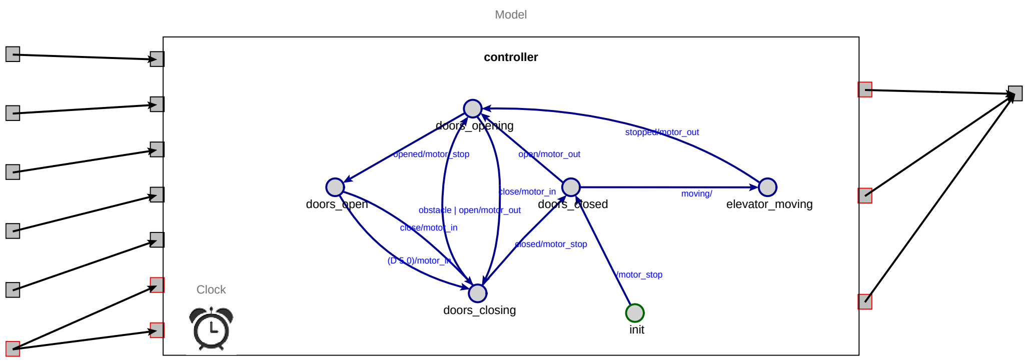


Figure 4.6: The TFSM model of an elevator door controller in *AToM*^{3}.

A Model of the Elevator Door

The main model of the elevator door is shown in Figure 4.5. A Model is visualised as a large rectangle, with its clock in the lower left corner. Inside, its Block elements are visualised as white squares, with input Pins on the left and output Pins on the right. The Relation between output Pins and input Pins is visualised by arrows. As usual in *ModHel'X*, the main model is a Discrete Event (DE) model. This will allow for a more optimal simulation of the model, as a DE model of computation only reacts when something happens. Intermediate time points are skipped. Because DE is typically used as an outer model, blocks like scenario to specify a scenario similar to an input model of Chapter 2, and plotter and display to show output traces similar to visualising output models in Chapter 2. In between the actual model consists of a feedback loop between the controller and a plant model, with a delay in between in order to avoid algebraic loops. The controllerAB block is an adaptation block responsible for the semantic adaptation of DE to TFSM, and doorAB is an adaptation block for the semantic adaptation of DE to SDF. In conclusion, the controller and plant model communicate through an outer DE model representing a communications bus between door and button panel. The bus is not modelled as a component of the system for the sake of simplicity, and is modelled as a medium for propagating discrete events instead. In [48] the approach presented in this section is however used to explicitly generate different bus architectures.

Outside of the DE Model, the simulation Engine exec is shown, with its two clocks Snapshot that keeps track of when new simulation steps need to be performed, and Stop that has the sole purpose to end the simulation when triggered. There is a FilteredImplication between Stop and Snapshot saying that Stop will be triggered after 10000 simulation steps. Additionally, the ASAPClock denotes that the simulation will happen *as soon as possible*, so not in real-time.

In the DSML, elements are created and attribute values are set through a dialog box when double clicking the element. In case of a AdaptationBlock, one of the dialog options is opening the inner model. The path to the *ModHel'X* API is set, so that all available BlockTypes, ModelOfComputations and AdaptationBlock types are collected. In the DSML, the type of each Block is subsequently chosen via a drop-down box listing all options.

Figure 4.6 shows the inner Model called controller of the controllerAB block, which models the elevator door controller. Instead of using a block diagram to model the TFSM, a different concrete

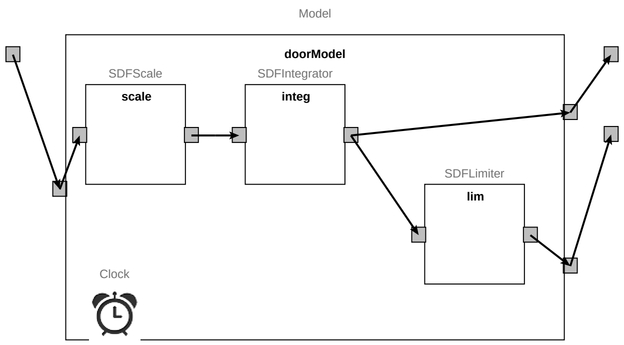


Figure 4.7: The DSF model of an elevator door plant model in *AToM*^{3}.

syntax is used that is more commonly interpreted as a state machine. The semantics are the same though, as the TFSM is transformed to a *ModHel'X* block diagram. There are six DE Pins at the left edge outside of controller and one on the right, corresponding to the input and output DE Pins of the controllerAB. They are connected to the input and from the output TFSM Pins of the controller, which represent simple TFSM input or output events that can occur. Data adaptation is done here by converting DE values (which can have any integer value) to TFSM values (which can be *true* or *false*) by setting properties on Pins. The five upper input TFSM Pins directly take over the values from the DE Pins that originally come from the scenario block, which are 0 or 1. The bottom DE Pin represents whether the door is fully open, fully closed, or neither, and is converted to two input events: open (if the DE Pin gets value 1, which is set as a properties entry) and close (if the DE Pin gets value 0). Input events are not subject to zero-order hold, meaning that they are not held afterwards if the condition is still valid, but they only occur when a value of the associated Pin changes. The three output Pins of controller are converted to a Pin representing the motor signal. The motor_out event (the door will go outward to open the door) is converted to a motor signal of 1, the motor_in event to -1 and the motor_stop event to 0. Similarly, the conversion is not subject to zero-order hold.

Figure 4.7 shows the plant model inside doorAB, which is executed every simulation step after controllerAB. The default block diagram syntax is once again used, as it is a recognisable notation for SDF. The simple plant model consists of three Blocks to calculate the position of the door:

— scale scales the motor input according to the step size, to obtain a realistic door movement speed;

— integ sums up the door position increases and decreases by the motor to obtain the new position;

— lim models whether the door is fully open or closed, which will be directed through a Delay block back to the controller. lim returns 1 if the position value is at least 5 (fully open), -1 if the position value is at most 0 (fully closed), and 0 if the position value is in between these two values.

The SDF Pin values can be directly forward from/to the doorAB DE Pins. Furthermore, the step size can be set in the adaptation block as a properties entry. As a result, the global model of the elevator door is a heterogeneous model involving three different modelling paradigms: DE, TFSM and SDF. The semantic adaptation of the adaptation block between DE and SDF is more complex,

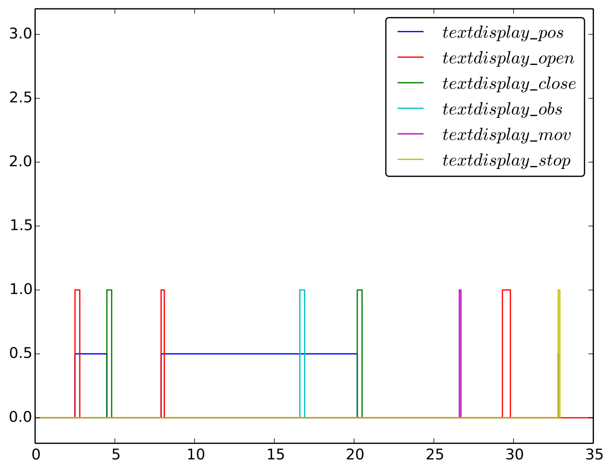


Figure 4.8: The simulation trace without semantic adaptation.

and is the subject of the next section.

4.3.4 A DSML and Execution Semantics for Semantic Adaptation

If the example is executed with standard adaptation blocks (such as presented in *e.g.,* Ptolemy II [56]), the behaviour is not as expected: the plant model is not sampled periodically, meaning that the door is not opening or closing with a constant speed. This is because it receives control each time the controller sends a command to it, and it has therefore no proper dynamics. The result is shown in Figure 4.8, where a large time step is chosen to clearly show the effect of a lack of semantic adaptation. Whenever a button is pressed (*i.e.,* when its signal is 1), the SDF model only gets control for one time step and the door is opened or closed with the according distance (in this case, 0.5). This time step may be at an irregular (*i.e.,* not according to its periodic clock) moment. Clearly, there is a need for semantic adaptation, more specifically the adaptation of data, control and time in an adaptation block.

The DSML for semantic adaptation provides the modeller with the concepts that are necessary for specifying the glue between heterogeneous models. By insulating the modeller from platform specific issues, it allows him to focus on the problem at hand, and it also reduces the semantic gap between what should be specified and how it is realised. A DSML implementing this metamodel, the presented semantics and a concrete textual syntax are defined in metaDepth [42]. The solution includes an ANTLR-based parser, an EOL script for generating code for the rules, and an EGL script [43] for generating Java code for *ModHel'X*. These steps ensure that the transformation from DSML code to Java code is entirely automated.

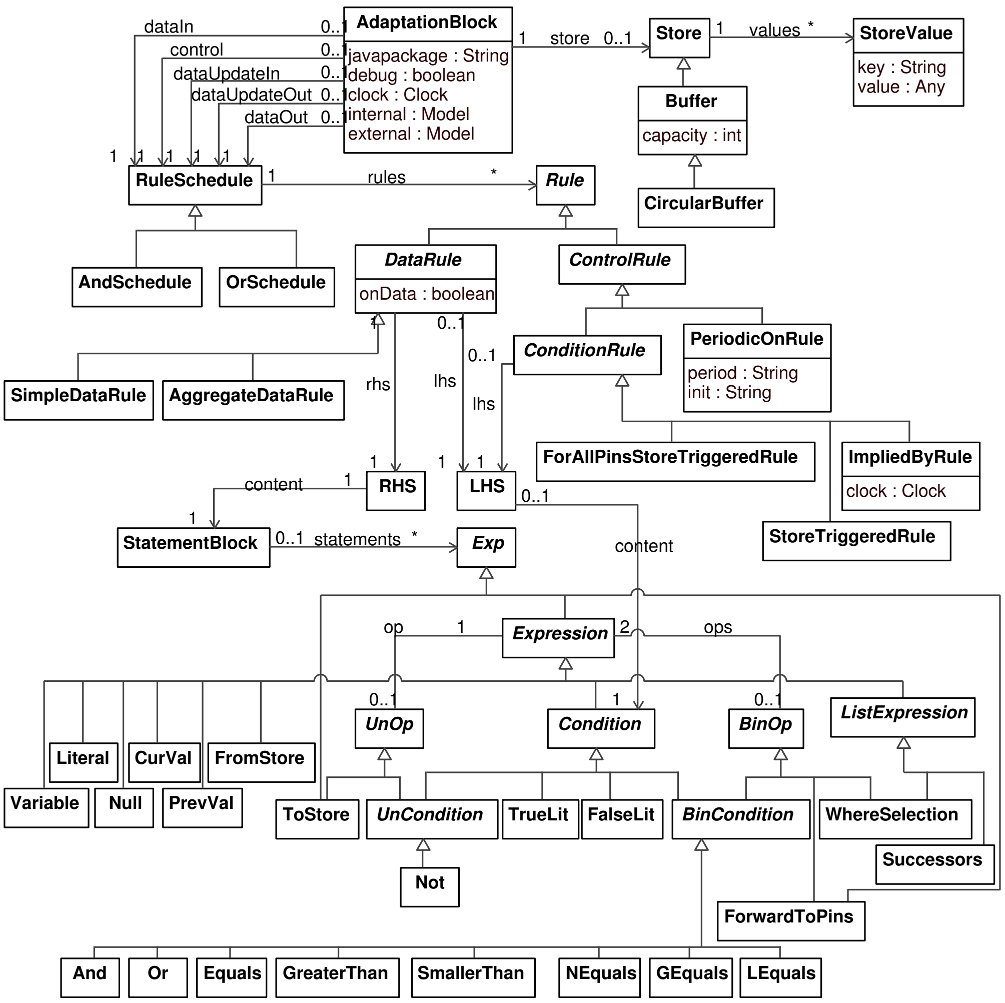


Figure 4.9: Extension of AdaptationBlock in the *ModHel'X* metamodel.

The Adaptation Block Metamodel

Figure 4.9 shows an extension of the *ModHel'X* metamodel of Figure 4.4 where the AdaptationBlock is decomposed. Through the input Pins of the AdaptationBlock, data Tokens can be received during simulation from the parent Model, and if needed saved in a Store. The output Pins transmit data to the parent Model. The Store explicitly models the state of the AdaptationBlock and can be a very simple data structure to save the current Token on an input Pin but also a complex Buffer to store multiple previous Token of an input Pin. Besides the Store, the AdaptationBlock can also register a new Clock in the Engine. This Clock can be used to request observation of the embedded Model at different time instances during execution.

The DSML for describing the behaviour of an adaptation block is based on Rules, which are evaluated and return *true* when they match. DataRules and ControlRules are used to match conditions for the semantic adaptation of data and control. The order in which Rules are evaluated is controlled by a RuleSchedule. Two different scheduling policies are defined: the AndSchedule evaluates the rules in order as long as they return *true*, and it returns *true* if all rules returned *true*; the OrSchedule evaluates the rules in order until one returns *true*, in which case *true* is returned. The return value is only important for ControlRules, as it decides whether control must be given to the embedded model (see also below for a description of the execution semantics).

DataRules are responsible for the adaptation of the data from the parent Model to the embedded Model and vice versa. DataRules have a left-hand side (LHS), denoting pre-condition for the applicability of the rule and a right-hand side (RHS) providing the needed action when the LHS is matched. DataRules have access to Pins and a Store of the AdaptationBlock which serves as a map of variables (variable key - value pairs). ControlRules are responsible for executing the embedded model (*i.e.,* giving control) at the current time instant. ControlRules have access to the Store to create complex trigger events but they also access the AdaptationBlock Clock to create an observation request in the future.

To ease the creation of Rules, we identified some common rule patterns, shown as subclasses of DataRule and ControlRule in Figure 4.9:

- SimpleDataRule: this data rule is evaluated for every parameter Pin. If its LHS evaluates to *true*, its RHS is executed. If onData is *true*, the RHS can only be executed if there are new Tokens on the Pins;
- AggregateDataRule: this data rule allows for more complex patterns over multiple parameter Pins, and will only be evaluated once. If its LHS evaluates to *true*, its RHS is executed;
- PeriodicOnRule: this control rule periodically gives control to the AdaptationBlock, and variable names are given for period and offset so that they can be set when the AdaptationBlock is used in a Model. The clock calculus in *ModHel'X* will make sure that the AdaptationBlock is updated at the specified moments;
- StoreTriggeredRule: this control rule gives control to the embedded Model if its LHS (with access to the Store) evaluates to *true*;
- ForAllPinsStoreTriggeredRule: same as a StoreTriggeredRule, but the LHS must be valid for each Pin;
- ImpliedByRule: this control rule gives control to the embedded Model whenever a given Clock is triggered and its LHS evaluates to *true*. This Clock can be triggered by ImplicationRelations.

This concludes the support for adaptation of data and control. For the semantic adaptation of time, an AdaptationBlock can contain TagRelations between Clocks, which allows to convert dates between their time scales.

The remainder of the DSML is composed of a small programming language consisting of expressions that can be used in the LHS and RHS of Rules. In contrast to the Java code of *ModHel'X*, the DSML includes domain-specific primitives. Many domain-specific expressions are included to improve the intuitive notation of the adaptation block. The language is not considered complete, and can be extended with more kinds of expressions if necessary. All implemented expressions are listed below with a short explanation, and if necessary, an example showing the concrete syntax:

— Exp: a statement or expression that does not necessarily have a return value;

— Expression: an expression with a return value;

— StatementBlock: a block of Exp statements used as action in an RHS;

— Condition: an expression with a boolean return value, that can be used as LHS;

— ListExpression: an expression with a list return value;

— UnOp: a unary operation that has one Expression operand;

— BinOp: a binary operation that has two Expression operands;

— Variable: returns the value of a floating point variable that is external of the rule;

— Literal: returns a literal floating point value;

— Null: returns the null value. Expressed as `NULL`;

— CurVal: returns the Token value of the context Pin. The context Pin is the Pin that is the current Pin in the lexical scope. Expressed as `VALUE`;

— PrevVal: returns the previous Token value of the context Pin, which is implicitly stored in the Store. Can be used to *e.g.,* see whether a token value has increased or decreased. Expressed as `PREVVAL`;

— FromStore: returns the value from the Store that has the context Pin as key. Expressed as `FROMSTORE`;

— ToStore: stores a value in the Store with Pin as key. Expressed as `TOSTORE`;

— Not, And, Or, Equals, GreaterThan, SmallerThan, NEquals, GEquals, LEquals: returns the result of the corresponding operation;

— TrueLit: returns *true*. Expressed as `TRUE` or `ALWAYS`, so that it can serve as an empty LHS;

— FalseLit: returns *false*. Expressed as `FALSE`;

— ForwardToPins: forwards the value given as first operand to the Pin(s) given in the second operand. Expressed as `Forward <op1> TO <op2>`;

— Successors: returns all Pins connected to the context Pin. Expressed as `SUCCESSORS`;

— WhereSelection: returns all elements of the first list parameter, for which the second condition expression is *true* (with an iterator as context). Expressed as `SELECT <op1> WHERE <op2>`;

Execution Semantics in Five Phases

As shown in Figure 4.9, the adaptation block has five rule schedules for adapting data and control. The semantics of these five different phases are shown in Figure 4.10. Each phase has its own schedule (containing a set of rules), and each phase is provided with a set of pins as parameters so that, besides the store, token values on pins become accessible in the rules. In case of the Data-in, Control-rules and Data-update-in phases, parameter pins are the input pins of the adaptation block. In case of the Data-update-out or Data-out phases the parameter pins are the output pins of the embedded model. The phases are executed in the following order when control has been given to the adaptation block:

— Data-in (data rules): when the adaptation block receives control, rules are executed to update

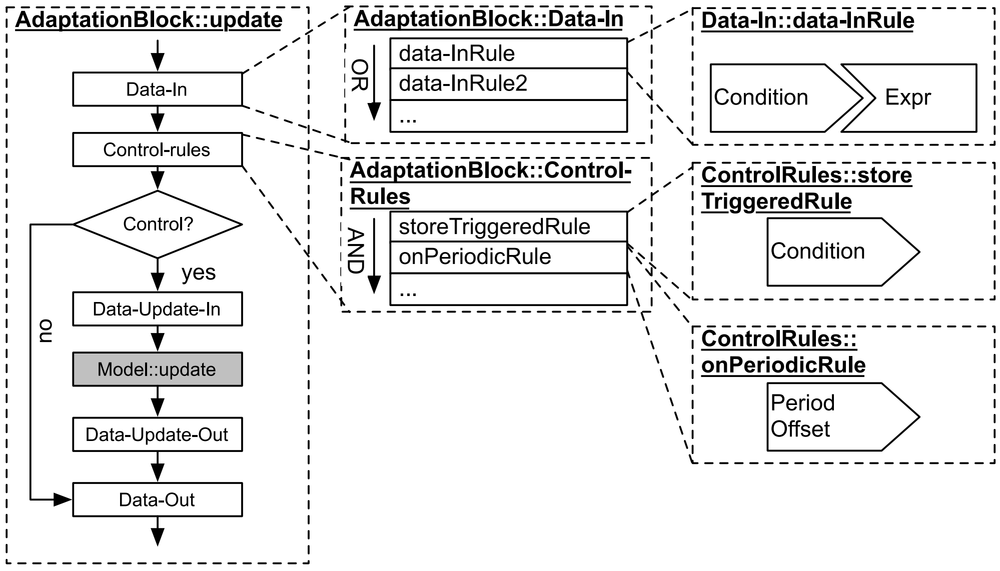


Figure 4.10: Execution of an adaptation block.

the store of the block according to the incoming data tokens. More complex operations (*e.g.,* calculating an average of all token values) transform the data for further processing;

— Control (control rules): control rules decide if control should be passed to the embedded model at the current instant according to the store of the block and the data on the input pins. Control rules also create new observation request on the clock of the adaptation block so that it receives control later (*e.g.,* to emulate a timeout);

— Data-update-in (data rules): if the control rules give control to the embedded model, the Data-update-in schedule is executed before control is passed to the embedded model. These rules provide the internal model with correct inputs according to the values in the store and the data on the input pins of the adaptation block. These rules are not evaluated if control is not given to the embedded model;

— Data-update-out (data rules): after the execution of the embedded model, the Data-update-out rules are used to update the store with the data available on the output pins of the embedded model;

— Data-out (data rules): finally, the Data-out rules are in charge of producing the outputs of the adaptation block from the data available in the store. This phase is executed even when control is not given to the embedded model because the adaptation block may have to produce some outputs any way (*e.g.,* in case of zero-order hold).

```
 1 operation ForwardToPins compile() : String {
 2         return "forwardToPins(" + self.op1.compile() + ", " + self.op2.compile() + ")"; }
 3 operation Successors    compile() : String { return "p.getSuccessorPins()"; }
 4 operation CurVal        compile() : String { return "cur_value"; }
 5 operation PrevVal       compile() : String { return "prev_value"; }
 6 operation TrueLit       compile() : String { return "true"; }
 7 operation ToStore       compile() : String { return "toStore(p, " + self.op1.compile() + ")"; }
 8 operation FromStore     compile() : String { return "fromStore(p)"; }
 9 operation NEquals       compile() : String {
10         return "!" + self.op1.compile() + ".equals(" + self.op2.compile() + ")"; }
11 // ...
```

Listing 4.21: Selected operations for compiling Exps.

Compilation to the Java *ModHel'X* API

The compilation to Java is performed in two steps, and can be implemented relatively straightforward because of the strict DSML structure. In a preprocessing phase, all Exps are compiled to Java code snippets using EOL. The second phase is the actual code generation phase, written in EGL, where the complete Java code is generated from the entire model.

The first phase is partly shown in Listing 4.21. The LHS and RHS Exp structures are transformed to strings of Java code and added as fields to the LHS and RHS elements, called respectively lhsstring and rhsstring. The code relies on name-based conventions, such as using `cur_value` and `prev_value` (lines 4–5), as well as a generic API, partly from *ModHel'X* itself (*e.g.,* the `getSuccessorPins` method on line 3), and partly specifically implemented for the DSML (*e.g.,* the `forwardToPins` method on lines 1–2 and the `toStore` and `fromStore` methods on lines 7–8).

In Listing 4.22 the compilation of the data-in phase (lines 1–7), an OrSchedule (lines 9–23), and SimpleDataRules (lines 25–45) are shown.

The data-in phase overrides a *ModHel'X* AdaptationBlock method called `adaptIn` (lines 1–2) which is automatically called during simulation. If a RuleSchedule is set for the data-in phase, the respective private method is called (lines 4–6).

An OrSchedule is compiled to a private method with a Boolean return value (lines 9–10). If the schedule contains no rules, *true* is set as return value (lines 12–14). Then, a call to the private method implementing each of the schedule's rule is made (lines 15–16), and when one of them evaluates to *true*, the return value of the schedule is set to *true* (line 17). A line is printed to the console if the debug field in AdaptationBlock is set to *true* (line 20), and the return value is returned (line 21).

Rules are compiled to private methods (lines 25–26) with the respective (as explained above) Pins as parameter. Then, the rule is evaluated for every Pin. In case the onData field in the Rule is *true*, an additional check for Tokens is performed (lines 29–31). Next, the `cur_value` and `prev_value` variables are set, so that CurVal and PrevVal expressions can be used in LHS and RHS (lines 32–35). The `cur_value` value is retrieved from the Pin Token, the `prev_value` value is retrieved from the Store. Next the LHS is evaluated (lines 36–37) and if it returns *true*, the RHS is executed (line 38). Because a DataRule's return value is not relevant (only ControlRule return values influence the AdaptationBlock behaviour), the method simply returns *true* (line 44).

Other rules are compiled similarly, and necessary generic helper functions (*e.g.,* `fromStore`) are generated during compilation. Additionally, the fields of AdaptationBlock are compiled to set up the *ModHel'X* AdaptationBlock adequately.

Listing 4.22: The EGL code for compiling the data-in phase, an OrSchedule and SimpleDataRules.

```
@Override
public void adaptIn() {
  // Update input values when events occur
  [% if (IB.datarulesin.asString() <> '') { %]
  [%=IB.datarulesin %](this.getInputPins());
  [% } %]
}
// ...
[% for (schedule in OrSchedule.allInstances()) { %]
private boolean [%=schedule %](Collection<Pin> pins) {
  boolean returnvalue = false;
  [% if (schedule.rules.size() = 0) { %]
  returnvalue = true;
  [% } %]
  [% for (rule in schedule.rules) { %]
  if ([%=rule %](pins)) {
    returnvalue = true;
  }
  [% } %]
  if (debug) System.out.println(" "+"Schedule [%=schedule %] returning " + returnvalue);
  return returnvalue;
}
[% } %]
// ...
[% for (rule in SimpleDataRule.allInstances()) { %]
private boolean [%=rule %](Collection<Pin> pins) {
  if (debug) System.out.println(" "+"SimpleDataRule [%=rule %]");
  for (Pin p : pins) {
    [% if (rule.ondata) { %]
    while (p.hasToken()) {
    [% } %]
      Object cur_value = null;
      try { cur_value = p.getToken(true).getValue(); } catch (IndexOutOfBoundsException e) {}
      Object prev_value = null;
      try { prev_value = this.fromStore(p); } catch (IndexOutOfBoundsException e) {}
      boolean condition = [%=rule.lhsstring/*.replace("\\n", "\n").replace("\'", "\"")*/ %];
      if (condition) {
        [%=rule.rhsstring/*.replace("\\n", "\n").replace("\'", "\"")*/ %]
      }
    [% if (rule.ondata) { %]
    }
    [% } %]
  }
  return true;
}
[% } %]
```

4.3.5 A Semantic Adaptation DSML in Practice

In this section, we reconstruct the default composition behaviour that results in the simulation trace of Figure 4.8 and the correct semantic adaptation between DE and SDF for the elevator door. The models presented in this section are transformed using the *metaDepth* framework into *ModHel'X* models in Java code.

Listing 4.23 and Listing 4.24 show adaptation blocks for both behaviours as a textual DSML model. The first 8 lines state the structural properties of the adaptation block and are the same for both behaviours. Line 1 presents the model name. On lines 2–3, the Java class and package are specified for the code generator. Line 4 presents the clock of the adaptation block. In this case the adaptation block has a timed clock, meaning that events occur at specific dates on a time scale (as opposed to an untimed clock, where events have no date). Lines 5–6 and 7–8 respectively present

```
 1 AdaptationBlock {
 2   IMPLEMENTS "DE_SDF_AdaptationBlock"
 3           IN "tests.elevatordoor"
 4   TIMED CLOCK abClock
 5   EXTERNAL MODEL de (
 6     "AbstractDEMoC" WITH TIMED CLOCK deClock)
 7   INTERNAL MODEL sdf (
 8     "SDFMoC" WITH UNTIMED CLOCK sdfClock)
 9   RULES:
10     IN
11     CONTROL
12     UPDATEIN
13       FORALLPINS ALWAYS ->
14                   FORWARD VALUE TO SUCCESSORS;
15     UPDATEOUT
16       FORALLPINS ALWAYS ->
17                   FORWARD VALUE TO SUCCESSORS;
18     OUT
19 }
```

Listing 4.23: Model of the default Ptolemy II behaviour

```
 1 AdaptationBlock {
 2   IMPLEMENTS "DE_SDF_AdaptationBlock"
 3           IN "tests.elevatordoor"
 4   TIMED CLOCK abClock
 5   EXTERNAL MODEL de (
 6     "AbstractDEMoC" WITH TIMED CLOCK deClock)
 7   INTERNAL MODEL sdf (
 8     "SDFMoC" WITH UNTIMED CLOCK sdfClock)
 9   RULES:
10     IN
11       FORALLPINS ON DATA: ALWAYS -> TOSTORE(VALUE);
12     CONTROL
13       PERIODIC AT "init_time" EVERY "period"
14     UPDATEIN
15       FORALLPINS ALWAYS ->
16                   FORWARD FROMSTORE TO SUCCESSORS;
17     UPDATEOUT
18       FORALLPINS ON DATA:
19         VALUE != FROMSTORE ->
20                   TOSTORE(VALUE);
21                   FORWARD VALUE TO SUCCESSORS;
22     OUT
23   TAG RELATION deClock = abClock
24 }
```

Listing 4.24: Model of the adaptation block with semantic adaptation

the external (parent) model and internal (embedded) model. The parent model is called `de` and adheres the *ModHel'X* `AbstractDEMoC` with a timed clock named `deClock`. Similarly, the embedded model uses SDF, which is by nature untimed.

From line 9 onward, the rules that model the execution semantics are specified. These differ in both models. The five rule phases, in their execution order, can be recognised from line 10 onward.

The default composition behaviour of Listing 4.23 does little semantic adaptation. No particular Data-in rules are given, since the store is not used. There is no Control rule, because whenever the adaptation block receives control, it will immediately give control to the embedded model. On Data-update-in, the tokens on the input ports are forwarded with no adaptation to the embedded model. This is modelled as the data rule on lines 13–14, which should be read as *LHS* $\rightarrow$ *RHS*. As a SimpleDataRule (denoted by `FORALLPINS`, EGL compilation code shown in Listing 4.22), it is executed for every input pin of the adaptation block. The LHS is `ALWAYS`, as there are no restrictions on when to forward data. The RHS specifies that the current token value CurVal (`VALUE`) should be forwarded to the embedded model's input pins to which the adaptation block input pins are connected to (`SUCCESSORS`). In the Data-update-out phase on lines 16–17, the tokens generated by the embedded model are similarly forwarded in another SimpleDataRule. As mentioned before, in this phase `VALUE` and `SUCCESSORS` – which depend on the parameter pins – refer to the token values and the successors of the embedded model's output pins. In summary, the default composition behaviour forwards all data instantaneously.

The intended behaviour with semantic adaptation of the DE to SDF adaptation block is shown in Listing 4.24. Its simulation trace is shown in Figure 4.11, where the door steadily goes up and down according to the scenario. The correct behaviour is more distinguishable when focusing on a stop event around 33 seconds with large SDF steps of 0.5 seconds, as shown in Figure 4.12. The adaptation block delays giving control to the SDF model until a new time step has started.

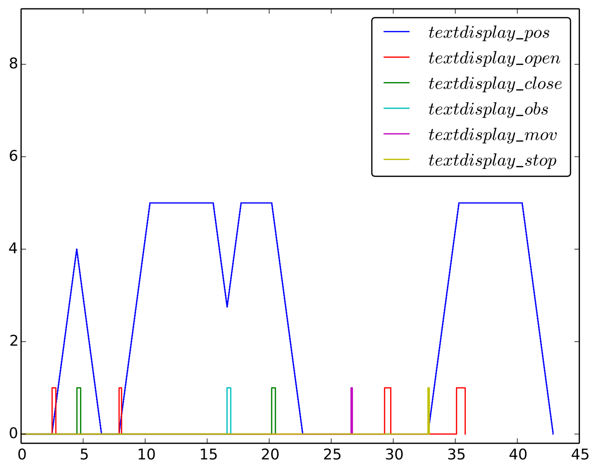


Figure 4.11: The simulation trace with semantic adaptation.

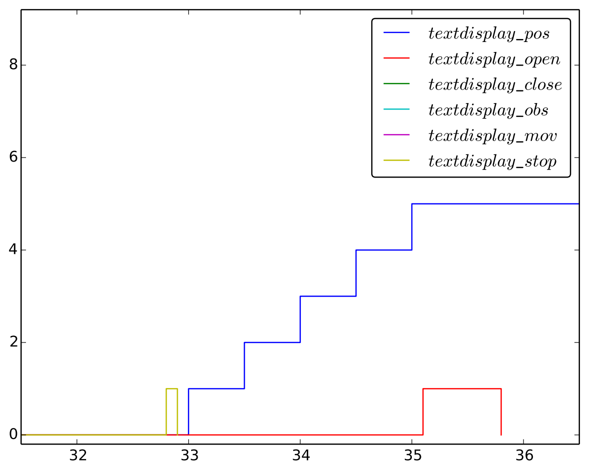
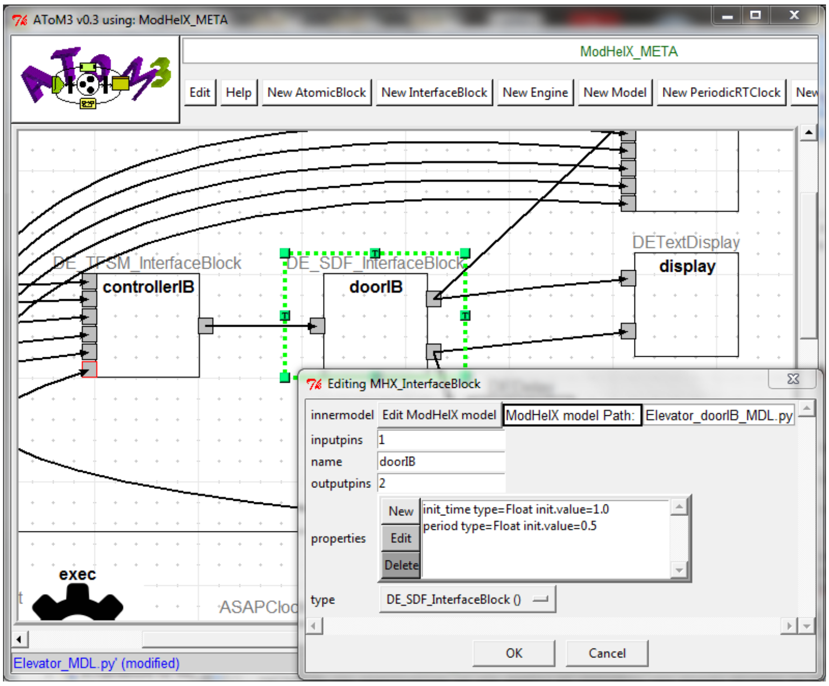


Figure 4.12: A part of the simulation trace with semantic adaptation with a very large SDF step size.

The adaptation block periodically samples the SDF model. Data is provided to the embedded model by applying zero-order hold from the input pins of the parent model to the input pins of the embedded model, meaning that the last known value should be used. The Data-in rule on line 11 saves the data from the input pins in the store, each time a token is received (`ON DATA`). The PeriodicOnRule on line 13 specifies the periodic sampling of the embedded model, starting at time "init_time", with a step of "period", which can be set in the instance model. The Data-update-in



innermodel: Edit ModHelX model | ModHelX model Path: | Elevator_doorIB_MDL.py
inputpins: 1
name: doorIB
outputpins: 2
properties:
init_time type=Float init.value=1.0
period type=Float init.value=0.5
type: DE_SDF_InterfaceBlock ()

Figure 4.13: An *AToM*^{3} screenshot from showing how a generated AdaptationBlock can be set as BlockType.

rule on lines 15–16 implements the zero-order hold functionality by providing the input pins of the embedded model with data from the store. Note that first-order hold (linear extrapolation) could also be modelled by calculating new values based on the previous ones (stored the store) every time the adaptation block is updated. A Data-update-out SimpleDataRule rule (lines 18–21) checks for every pin whether the embedded model's output value changed, by comparing it with the previous one, which was stored. Initially the value in the store is *null* making sure the LHS evaluates to *true*. Only when the output of the embedded model changes, tokens are forwarded to the parent model. No event is consequently produced when the window is inactive. Finally a SameTagRelation on line 23 specifies that time is measured the same way in the adaptation block and in the embedding model. Note that in both models rules never had to explicitly scheduled, as every phase contained at most a single rule.

4.3.6 Discussion

The DSML for heterogeneous modelling allows one to graphically model generic models, and assign a different model of computation to every model. The DSML for semantic adaptation allows one to explicitly specify the adaptation behaviour intended at the boundary between two heterogeneous parts of a model. Both DSMLs address a different part of a heterogeneous system, and are meant to be used together. First, an AdaptationBlock is specified using textual syntax such as in Listing 4.24, and a Java class (such as `DE_SDF_AdaptationBlock`) is generated. Then, a heterogeneous model is created in *AToM*^{3} such as Figure 4.5, with an AdaptationBlock instance for which the BlockType is set to the previously generated AdaptationBlock as shown in the *AToM*^{3} screenshot of Figure 4.13.

The use of both DSMLs is an improvement in three respects. First, models are created in a graphical tool, which makes the topology of the models clear. Second, semantic adaptation is clearly stated and configurable. Third, our approach leaves the models unchanged. This improves modularity, as a given model may be embedded as is in different contexts; what needs to be changed each time is just the adaptation layer defined using our DSML.

The proposed DSML was used to model the adaptation of the DE model to the TFSM model of the elevator door system. The DSML is able to model all the needed constructs for the adaptation block and generate Java code. Some constructs necessary for the adaptation of a wide variety of models of computation may not be available in the DSML for semantic adaptation yet. Nevertheless, the DSML offers a solid base, both syntactically and semantically, to which new language constructs can be added.

The improvements mentioned here (specifying adaptation between models explicitly while keeping the modularity of models) have been available in *ModHel'X* for a few years. However in *ModHel'X* adaptation is coded in Java, so there is a semantic gap between high-level adaptation mechanisms and the way it is actually written as low-level code that makes minute manipulations of data structures. Moreover, an adaptation block in *ModHel'X* is scattered in several methods, which makes its behaviour hard to follow. The five phased execution semantics and the DSML are focused and concise: they bridge this semantic gap. More than 300 lines of Java code are reduced to less than 25 lines of DSML code.

4.4 Related Work

According to the structure of this chapter, we split up related work in two parts: composition of modelling languages, and heterogeneous modelling and semantic adaptation.

4.4.1 Composition of Modelling Languages

In [181], the authors propose a language embedding technique for creating families of DSMLs. The abstract syntax is defined through metamodelling, the textual concrete syntax is specified by means of a mapping between metaelements and keywords, and the translational semantics is expressed by means of a proper transformation language. All the methodology relies on a "host" language (in this case Ruby) that has to be expressive enough to allow the definition of keyword extensions, scope restrictions, type and operator extensions, and overrides. A similar approach is proposed in [156] by using the ATLAS Transformation Language (ATL) to combine DSMLs semantics. The main distinction with the technique discussed in this work is the expressive power of the template/binding mechanisms, which allow us to obtain extensive forms of language combinations. In contrast, in [181, 156], merging and importing methods are exploited to address modularity.

Other approaches provide host language mechanisms to support composition at the programming language level of abstraction [60, 211]. While they succeed in composing abstract and concrete syntaxes, they typically face semantics composition issues.

In [202], the author discusses mechanisms for combining DSMLs with a particular focus on multiview-based modelling approaches. A similar approach in [52] combines software architecture DSMLs. DSMLs can be combined in several interesting ways by drawing correspondences among different languages, though these approaches do not offer combination facilities comparable to the template/binding mechanism illustrated in this work. Semantics combination is addressed only in [52] and limited to the software architecture domain. Moreover, in general, merging graphical concrete syntaxes discloses additional and non-trivial issues inherent to visual languages and implicit semantics associated to symbols [52, 202]. That problem does not occur in our work because of the exploitation of textual concrete syntaxes.

The work in [213] proposes to exploit feature diagrams to combine and refine DSMLs in a product line approach fashion. Feature diagrams allow to select the set of features to be imported by each language. While such technique is intuitive from the syntax points of view, it is not clear how semantics can be combined.

The use of templates in modelling is not new. They are already present in the UML 2.0 specification [153], as well as in approaches like Catalysis' model frameworks [54] and package templates, and the aspect-oriented metamodelling approach of [33]. Interestingly, while all of them consider templates for metamodels or class diagrams, none consider concepts or model templates. Moreover, they do not consider concrete syntax or semantics.

In chapter 4 of his thesis [126], Mannadiar presents a template-based approach for the design of modelling languages. The three aspects of a language, namely abstract syntax, (visual) concrete syntax and (translational) semantics. The combination of semantics is restricted to discrete-event semantics.

During the course of his master's thesis at the University of Antwerp, Ugaz presented a survey of composition of modelling languages [199]. Composition techniques in modelling, such as aspect-oriented modelling, are extensively discussed, as well as approaches that combine visual concrete syntax. The survey distinguishes eleven composition techniques, and subsequently, these techniques are analysed. Most of the composition techniques are homogeneous, as they combine models in the same language. Language composition in *metaDepth* is homogeneous, semantic adaptation is heterogeneous because different models of computation can be combined (although it can be considered syntactically homogeneous as well). The only other technique that is heterogeneous is interfacing [59]. The number of generic and specific composition techniques are evenly spread, and many techniques focus on the composition of UML models [153, 98, 214], and more specifically of Class Diagrams [115, 59, 155, 167]. We focus on generic techniques that can be applied to a wide variety of DSMLs. Other techniques also provide a generic solution [164, 10, 202, 144, 63]. It is concluded that semantic adaptation is the only technique that supports continuous-time models. Furthermore, the composition of modelling languages in *metaDepth* is the only approach that is implemented and that addresses all three components of a language (abstract syntax, concrete syntax and semantics). Interestingly, Pedro *et al.* use the parametrisation technique to combine visual concrete syntax [157]. According to the survey, the techniques used in this chapter are unique in the sense that the actual composition is done at run-time. The reason for this is that we focus on the combination of semantics, which we consider to be operational in this chapter. The approaches presented in this chapter require manual composition, and are considered highly modular. We focus on manual composition to enhance the expressive power of the language engineer. In contrast, a number of approaches define an automatic composition [153, 10, 101, 167, 63]. Unlike our approach, where it is assumed that there are no conflicts, other approaches include conflict resolution [164, 63] or avoidance [167, 38, 101].

4.4.2 Heterogeneous Modelling and Semantic Adaptation

To focus of the related work for semantic adaptation is on three different tools for heterogeneous modelling and simulation: Ptolemy II [56], Simulink®/Stateflow®[2] and ModHel'X [23]. All of these support heterogeneous modelling and they all use hierarchy as a mechanism for composing the heterogeneous parts of a model. Other types of approaches are described in detail in [81].

In Simulink®/Stateflow®, a power window system which has many similarities compared to the elevator door, has been studied at different levels of detail. The resulting models are available as demos in the tool and became the seminal case study for semantic adaptation[3]. Consequently the power window system is also modelled using Ptolemy II and *ModHel'X*, and the resulting models are available online[4].

Ptolemy II also explicitly supports the principle of models of computation. However, Ptolemy II does not provide a default semantic adaptation between heterogeneous parts of a model. In the power window example, the Window component (analogue to the elevator door plant model), which is modelled using SDF, is interpreted as a DE component by the DE director. It is therefore activated each time it receives an input event, and each data sample it produces is considered as a DE event on its outputs. This results in a similar simulation trace as shown in Figure 4.8.

In order to model this semantic adaptation, the main DE model has to be modified in a way that depends on the details of the realisation of the window model. If another modelling paradigm is to be used for the plant model however, this adaptation would have to be changed. However, the modification depends on the internals of the window model, so the global model is not modular. Moreover, it is not easy to make a difference between a block which is part of the model and a block which is there for semantic adaptation. Semantic adaptation is therefore diffuse and difficult to specify and understand. We have successfully applied our approach to the power window case [135].

In Simulink®/Stateflow®, which is a powerful and efficient simulation tool for heterogeneous modelling, semantic adaptation is even more diffuse. The global semantics of a heterogeneous model (for instance a Simulink® model including a Stateflow® submodel) is given by one solver which is used to compute its execution. The solver uses parameters of the blocks and their ports, in particular a parameter called "sample time"[5], to determine when to compute the value of the different signals. These parameters can be considered a form of semantic adaptation of control and time. The resulting execution of the model depends on the value of these parameters and on the type and parameters of the solver itself. Even if default values for these parameters are determined by the simulation engine, the user must have a deep understanding of the solver's mechanisms and fine tune different simulation parameters to adapt the resulting execution semantics to her/his needs. A part of the semantic adaptation can also be performed using blocks like in Ptolemy II or

[2]http://www.mathworks.fr/products/simulink/

[3]See the online documentation at http://www.mathworks.fr/fr/help/simulink/examples/simulink-power-window-controller-specification.html

[4]Ptolemy II model: http://wwwdi.supelec.fr/software/downloads/ModHelX/power_window_fulladapt.moml
ModHel'X model: http://wwwdi.supelec.fr/software/ModHelX/PowerWindow

[5]http://www.mathworks.fr/fr/help/simulink/ug/types-of-sample-time.html

truth tables or functions, but with the same drawbacks. As a counterpart, the solvers of Simulink® are very efficient and configurable.

ModHel'X was presented in Section 4.3.1 and aims for modular semantic adaptation to overcome the shortcomings of Ptolemy II and Simulink®. An issue with *ModHel'X* is that the semantic adaptation is specified using calls to a Java API in different methods of an adaptation block. Therefore, constructing an adaptation block is tedious and error prone. The approach presented in Section 4.3 attempts to improve *ModHel'X* by providing DSMLs for modelling systems and modelling semantic adaptation. We used our approach to adapt hybrid languages that stem from legacy systems and are combined as black boxes. We use the Functional Mock-up Interface (FMI) standard for the communication between different language components over a bus. This allows users to reuse legacy simulators. In addition, the approach allows users to generate different bus architectures, depending on her/his needs.

4.5 Conclusion and Future Work

In this chapter we have shown how composition mechanisms extension, concepts and templates in *metaDepth* are used for composing languages. The three aspects of a modelling language, namely abstract syntax, concrete syntax and semantics, are combined. We have shown how there are clear correspondences between combining abstract syntax and concrete syntax. Combining semantics in *metaDepth* is done in a less structured way however, despite the use of hybrid concepts to explicitly model what operations are required. Different composition steps get mixed up while composing semantics, and more advanced semantic compositions lose all modularity.

These more advanced semantic compositions bring us to the concept of heterogeneous modelling and semantic adaptation. The heterogeneity of the elevator door was tackled by using models in a DSML with a visual concrete syntax and by assigning different models of computation. Semantic adaptation is the “glue” that gives well-defined semantics to the composition of heterogeneous models. A second DSML in *metaDepth* with textual syntax was introduces to bridge the cognitive gap between the implementation and specification of such semantic adaptation blocks. The model of an adaptation block explicitly specifies the adaptation of data, control and time using a set of rules. Furthermore, a model-to-text transformation is defined to generate the adaptation block code for the *ModHel'X* framework. The DSML enables the modeller to easily customise adaptation blocks that fit the heterogeneous models in a modular way, as involved models are left untouched. The approach was illustrated on the elevator case study by reconstructing in *ModHel'X* both the implicit Ptolemy II adaptation and the specific one needed between DE and SDF to obtain the expected behaviour. We believe that the DSML and the execution semantics are, as a principle, expressive enough to model different semantic adaptations of heterogeneous models, though additional constructs might be needed for additional behaviours.

To summarise, we return to the reconsider the research questions stated in Section 4.1.

— *RQ 3.1.* How can abstract syntax be combined using these composition mechanisms?
We showed that abstract syntax can be composed by using extension to combine two metamodels and concepts to combine metamodels with templates in Section 4.2.3;

— *RQ 3.2.* How can textual concrete syntax be combined using these composition mechanisms?
We showed how to compose concrete syntax in a similar way to abstract syntax in Section 4.2.4.

We determined the need for support for inheritance in the specification of concrete syntax;

— *RQ 3.3.* How can operational semantics be combined using these composition mechanisms?
We showed that operational semantics can be composed by using operations, hybrid concepts and binding in Section 4.2.5. We observed however that composition of semantics is not strongly linked to composition of abstract syntax, so operational semantics cannot be composed as structured a way as is possible for abstract syntax;

— *RQ 3.4.* What are the assumptions and limitations of the approach?
In addition to the above-mentioned complications, we concluded in Section 4.2.6 that the composition mechanisms do not suffice for more complex composition of operational semantics. This further emphasises the need for a dedicated approach;

— *RQ 4.1.* What is an appropriate formalism for heterogeneous modelling?
We presented a visual DSML for heterogeneous modelling in *AToM*^{3} in Section 4.3.3, including a mapping to the *ModHel'X* Java framework for heterogeneous modelling. We were able to reuse a formalism in *AToM*^{3} for timed finite state machines. The visual syntax of the DSML better reflects the graph-like nature of the models;

— *RQ 4.2.* What is an appropriate formalism for semantic adaptation?
We described a textual DSML for semantic adaptation in *metaDepth* in Section 4.3.4. We enhanced the *ModHel'X* metamodel and introduced a rule-based language for defining the adaptation of data, control and time. Expressions in the DSML are highly specific to the domain of semantic adaptation;

— *RQ 4.3.* How is creating semantic adaptation improved by this formalism?
We were able to reconstruct the default Ptolemy II semantic adaptation of Discrete Event models to Synchronous Data Flow models, and corrected it with our own model in Section 4.3.5. The DSML models are only tens of lines of code, and are transformed to hundreds of lines of Java code;

— *RQ 4.4.* What are the assumptions and limitations of the approach?
In our approach, we assume a step-wise behaviour of models of computation. More specifically, a model of computation should be able to execute a finite step. This means that the necessary abstraction steps need to be made to map *e.g.,* continuous models to the approach. Successfully modelling semantic adaptation depends on the granularity of the involved models of computation, and the divisibility of its execution steps. Additionally, we do not guarantee the applicability of our DSML for semantic adaptation to any possible model of computation. To counter this, the DSML can be extended with new language constructs.

Future work is necessary to determine the exact boundaries of the composition mechanisms as presented in *metaDepth*. While it is clear from this chapter that language composition is possible, the question arises what the limitations are, and how modularity can be ensured. Also, some slight adaptations can be done to *metaDepth* to alleviate the minor inconveniences for *e.g.,* composing concrete syntax. Additionally it remains to be seen whether these techniques are also applicable to a visual concrete syntax. Rafael Ugaz explored this topic in his master thesis at the University of Antwerp [200]. His work also includes an extensive survey of existing techniques [199].

In the context of semantic adaptation, the models and semantic adaptation between them were explicitly modelled. What has been left implicit are the models of computation. By lack of explicit models of computation, they were considered as black boxes in this chapter. In future work, models of computation can be modelled explicitly. For this, a DSML could be again devised. This DSML can be based on a time calculus [22] to reason about simulation steps. In the context of this chapter, the main benefit of explicitly modelling models of computation is that constraints can be added extensively to the DSML for semantic adaptation to make sure that the semantics of adaptation blocks are correct. Additionally, an explicit model of computation also results in better modularity, reuse possibilities, analysis possibilities, platform independence, inter-tool operability and other benefits of DSM. Mustafiz *et al.* continue our work and combine models of computation that are represented as a combination of Statecharts and Class Diagrams [148]. In contrast with our work, their approach assumes that models of computation are white boxes.

A final remark is that when discussing the combination of semantics, we solely focused on operational semantics. Further research can be conducted to investigate how to compose languages with translational semantics. The combined language modules might map to the same semantic domain, or to different but possibly overlapping semantic domains. Related to this, composition of code generators of two language modules is interesting future work.

CHAPTER 5
Conclusion

5.1 Summary

The ever increasing complexity in systems we analyse and design results in the rise of Model-Driven Engineering (MDE) [182]. Such complex systems require solutions for various disparate problems. This heterogeneity is tackled in *Multi-Paradigm Modelling* (MPM) by explicitly modelling all aspects of the system under study and modelling processes using the most appropriate modelling language(s), at the most appropriate level(s) of abstraction [146, 45]. The goal is to minimize "accidental" (as opposed to "essential") complexity [25] by capturing only the essence of a problem.

In Domain-Specific Modelling (DSM) [74] the general goal is to provide means for domain users to model systems in their problem domain. Techniques such as metamodelling and model transformation enable users to create Domain-Specific Modelling Languages (DSMLs) for domain users. Current DSM techniques allow users to model at the domain level and simulate the model, optimise the model, transform the model to other formalisms, test the model, synthesise code, generate documentation, etc. According to MPM principles, multiple aspects/views as well as subsystems are modelled using distinct DSMLs, which are specifically designed to address a specific facet of a problem. This implies that not only system modelling, but also modelling language engineering, becomes part of the development process. Modelling language engineering is thus a vital part of DSM. In this thesis, we assume that modelling languages are defined by a metamodel, a concrete syntax model and operational or translational semantics, all of which are explicitly modelled. This section summarises the three presented contributions to modelling language engineering, compatible with the MPM principles.

5.1.1 Verification for Domain-Specific Languages (Chapter 2)

Verifying whether a model satisfies a set of requirements is considered to be an important challenge in DSM [65]. It is nevertheless mostly neglected by current DSM approaches. Verification has been achieved by translating models to formal representations. Logic-based formulas in formalisms such as Linear Temporal Logic (LTL) [162] and Computation Tree Logic (CTL) [58] are used to represent the temporal properties that need to be verified [171]. These temporal properties can be verified using *e.g.,* model checking techniques [34]. Currently, domain users need to have a profound knowledge of some logic to express properties. This violates the principles of MPM. As with design models, the level of abstraction for specification and verification tasks needs to be raised to the domain level, as domain users should not be exposed to underlying notations and techniques [210]. In this sense, DSM should not only address modelling the design of a system, but also its properties, its environment, its run-time state, and its execution traces, which should all be modelled at the domain level, in their own DSML. No general DSM solution exists for specifying properties at a domain-specific level, matching the domain-specific level at which design models are specified. Furthermore, such domain-specific property specification requires the means to automatically verify whether a system design satisfies these properties. This led to the following research question:

RQ 1. Can we find a general MPM solution to support the specification of (temporal) properties, as well as their verification? This solution should include support for verifying these properties. The user should only be exposed to domain-specific concepts. The solution should weigh as little as possible on the lightweight DSM process, and should be an extension of the existing DSM techniques for design languages.

We presented a solution in the form of *ProMoBox*, a framework that integrates the definition and verification of temporal properties in discrete-time behavioural DSMLs, whose semantics can be described as a schedule of graph rewrite rules. Thanks to the expressiveness of graph rewriting, this covers a very large class of problems. With *ProMoBox*, the domain user models not only the system with a DSML, but also its properties, input model, run-time state and output trace. A DSML is thus comprised of five sublanguages, which share domain-specific syntax. The sublanguages are generated from a single metamodel to keep them consistent and to avoid duplication, that is annotated to denote the role of each language concept. The operational semantics of the DSML is modelled as a transformation and is annotated with information about input and output. The modelled system and its properties are transformed to *Promela*, and properties are verified with *Spin*, a tool for explicit state model checking. In case a counterexample is found, its execution trace is transformed to the domain-specific level as a trace model, which can be played out. Thus, whilst modelling and verifying properties, the domain user is shielded from underlying notations and techniques. Following MPM principles, the process of the *ProMoBox* framework is explicitly modelled in a Formalism Transformation Graph and Process Model.

We evaluated the *ProMoBox* approach in three ways. Firstly, we assessed the verification time in *Spin* of the generated *Promela* model, and anecdotally showed that execution times and memory consumption are no worse than their hand-made counterparts. Secondly, we showed that *ProMoBox* is a lightweight solution by comparing to existing DSM approaches. Thirdly, we investigated the impact of evolution of models in *ProMoBox*, to evaluate the approach's sensitivity to change. This included replacing the verification backbone of *Promela* and LTL by *Groove* and CTL. In conclusion, *ProMoBox* provides a solution for the specification and verification of properties in a highly flexible and automated way, according to MPM principles.

I was responsible for most of the ideas in this chapter. These ideas were shaped by collaborations with Manuel Wimmer (Vienna University of Technology). He introduced the idea of translating rule-based semantics to Promela. During this research, I also worked together with my advisor Hans Vangheluwe, Levi Lucio (McGill University), Eugene Syriani (Universié de Montréal), and Romuald Deshayes (Universié de Mons), who helped shape *ProMoBox*. Furthermore, I was responsible for the implementation and evaluation of *ProMoBox*.

5.1.2 Evolution of Domain-Specific Languages (Chapter 3)

In software engineering, the evolution of software artifacts is ubiquitous [131]. Diverse artifacts such as programs, data, requirements, and documentation may evolve. In DSM, where modelling languages play a central role, evolution occurs not only at the level of models, but also at the level of modelling languages. Language evolution applies in particular to DSMLs, where relatively frequent changes in the problem domain as well as in the implementation target domain (*e.g.,* due to external technical or strategic decisions) must be reflected in the respective languages. Possibly large numbers of modelling artifacts such as instance models or transformation models become invalid and unusable when a related DSML is modified/evolved. Consequently, support for (semi-)automated co-evolution of language artifacts is needed. Ad hoc approach are tedious and error-prone [197]. The reason for this is that syntax of languages such as UML [153] and BPMN [151], which have evolved considerably over the last few years, easily comprise several hundreds of elements. Also, the semantic differences resulting from this evolution, either intended or intentional, can be subtle. Hence, dealing with evolution requires in-depth knowledge of the language as a whole. Without a proper scientific foundation, as well as methods, techniques and tools to support evolution DSM cannot live up to its promise of ten-fold productivity increase [96]. This becomes apparent when projects span longer periods of time [180]. Most contributions in this field are focused on (semi-)automatic model differencing [30] and on the co-evolution of instance models [86]. Nevertheless, in a DSML relational ecosystem, there are other modelling artifacts that might have to co-evolve, which may in its turn trigger co-evolution of more modelling artifacts. In our work, we present a solution to the following research question:

RQ 2. What is a solution according to MPM principles for dealing with the consequences of language evolution?

We presented a framework that deals with all possible consequences of language evolution in a DSML relational ecosystem. In the framework, language evolution is de-constructed in a selection of possible delta operations, and associated migration operations to co-evolve instances, based on related work. Contrary to related approaches at the time of this work, we focus on the neglected problem of transformation co-evolution. We de-constructed the consequences of evolution and identified four basic evolution scenarios: model evolution, image evolution, domain evolution and transformation evolution, which can each be handled (semi-)automatically. Consequently, co-evolution of a transformation can be achieved by chaining, where necessary, inverse migration transformations before and migration transformations behind the transformation to make it applicable to evolved models. We described the projection problem that prevents the full automation of transformation model migration by use of the migration transformation for instance models. The projection problem occurs when the DSML semantically evolves (*i.e.,* its properties change), in contrast to refactoring. Nevertheless, by de-constructing language evolution, our approach isolates such manual interventions. Furthermore, when taking the DSML relational ecosystem into account, we showed that the co-evolution of related artifacts can be mapped onto these basic scenarios. Moreover, the basic evolution scenarios can be combined to address all complex language evolution scenarios. We identified a goal to satisfy consistency and continuity, meaning that the evolution must be syntactically correct (*i.e.,* the conformance relation is preserved throughout the system) and semantically correct (*i.e.,* the system has evolved according to the intended changes). These concepts were related to our framework.

Finally, we defined and implemented an approach for instance co-evolution and transformation co-evolution. We introduced a flexible and modular migration pipeline, that can be reused for instance model migration as well as transformation model migration. The pipeline offers a high degree of automation by maximally employing coupled migration operations and a high degree of control by isolating manual adaptation which is deliberately syntactically constrained. From these building blocks, we re-constructed a solution for transformation evolution. We distinguished three levels of automation and we explored all possible consequences of modelling language evolution in a feature diagram, and discussed the level of automation possible. We provided an algorithm that forms the backbone of our framework. The algorithm uses techniques that are introduced or discussed throughout the chapter and constitutes a coherent solution for the problem of modelling language evolution. The completeness of the framework is achieved by showing how artifacts and their relations can be mapped on a general DSML relational ecosystem. This is backed by the related work after the time of our research: we could not find any work that cannot be fit into the framework. We defined the technological prerequisites for maximally automating the consequences of language evolution.

This chapter presents a top-down approach to the problem of language evolution. We strove to make the chapter comprehensive, so this work can be used as a guideline for further research. This is demonstrated by related work done after the time of our research, which can be mapped to (parts of) our framework. For transformation co-evolution however, we present a specific approach and prototype implementation, as this was missing in existing approaches.

The work on evolution of modelling languages was my first topic as a Ph.D. candidate. The general idea for tackling transformation co-evolution, in particular the commuting diagram, was presented by my advisor Hans Vangheluwe. Further developments were the result of discussions with my advisor, and with Antonio Cicchetti (Mälardalen University), Manuel Wimmer (Vienna University of Technology) and Jonathan Sprinkle (University of Arizona), and of individual work. I was responsible for defining the framework and for the prototype in *metaDepth*.

5.1.3 Composition of Domain-Specific Languages (Chapter 4)

Techniques such as metamodelling and model transformation greatly facilitate the development of DSMLs. However, these methods for development require the language engineer to develop DSMLs from scratch. Nevertheless, there are similarities between DSMLs: one might need a new variant of Petri nets, a different kind of automaton, or the combination of a number of formalisms. Therefore, there is a clear need for reuse of existing languages, or language modules. In other domains, such as generic programming, a similar need for composition is addressed by the development of generic mechanisms [75]. We selected three composition mechanisms from the *metaDepth* tool, namely, extension, concepts and templates and investigated whether they are sufficient to allow composition of modelling languages in a structured way. All three concepts, abstract syntax, (textual) concrete syntax and (operational) semantics, of a modelling language need to be combined using these mechanisms. By using these mechanisms, we aim for general applicability of our approach, provided that these composition mechanisms are supported.

The combination of the operational semantics of languages can be very complex, but necessary in Multi-Paradigm Modelling. Languages can be semantically entangled, requiring a dedicated approach for the combination of semantics. These so-called heterogeneous systems must be executed correctly, while still maintaining modularity of languages, to enable reuse, and to allow the combination of languages as black boxes. A solution is to define “composition laws” for heterogeneous parts of a model, which have to be explicitly described. These composition laws are captured in a technique called *semantic adaptation* [23]. The authors developed *ModHel’X*, a tool for heterogeneous modelling and semantic adaptation. Semantic adaptation can be explicitly described separately from models and languages. Often complex semantic adaptations have to be coded however, which is tedious and error prone. Semantic adaptation too should be modelled explicitly, at the most appropriate level(s) of abstraction, in the most appropriate formalism(s). In this thesis, the work on composition of DSMLs is narrowed down to the following research questions:

RQ 3. Can we extend generic model composition mechanisms, inspired by generic programming, to the composition of DSMLs, in its three aspects, namely, abstract syntax, (textual) concrete syntax and (operational) semantics?

RQ 4. What is a solution according to MPM principles for dealing with execution of heterogeneous modelling and semantic adaptation?

We have shown how composition mechanisms extension, concepts and templates in *metaDepth* can be used for composing languages. The three aspects of a modelling language, namely abstract syntax, concrete syntax and semantics, are combined. Abstract syntax can be composed by using extension for combining two metamodels and concepts for combining metamodels with templates. We have shown that there are correspondences between combining abstract syntax and combining concrete syntax. We identified the need for support for inheritance in the specification of concrete syntax. Operational semantics can be composed by using (a) operations, (b) hybrid concepts to explicitly model what operations are required, and (c) binding of such concepts. In contrast to composition of concrete syntax, composition of semantics is not strongly linked to composition of abstract syntax. Different composition steps get mixed up while composing semantics, and more advanced semantic compositions lose all modularity. Consequently, using these composition mechanisms, operational semantics cannot be composed in a structured way. In addition to the above-mentioned complications, we concluded that the composition mechanisms do not suffice for more complex composition of operational semantics. This further emphasises the need for a dedicated approach for combining semantics. We illustrated the approach with a elevator door example, for which a Timed State Machine language is composed from a State Machine language, a language for Expressions, and a language modelling the concept of a Clock.

The need for more advanced semantic compositions was illustrated by the elevator door example. Reactive behaviour (*e.g.,* pressing buttons) and continuous-time behaviour (*e.g.,* door movement) are combined, but can be best modelled heterogeneously. This brought us to the concept of heterogeneous modelling and semantic adaptation. The heterogeneity of the elevator door was implemented as heterogeneous models that combine Discrete Event, Finite State Machine and Synchronous Data Flow models of computation. We introduced a DSML with a visual concrete syntax in *AToM^3* for heterogeneous modelling. Semantic adaptation acts as the “glue” between heterogeneous models, modelled as an adaptation block. We defined a second DSML in *metaDepth* with textual syntax to bridge the cognitive gap between the implementation and specification of such semantic adaptation blocks. The model of an adaptation block explicitly specifies the adaptation of data, control and time using a set of rules. Expressions in the DSML are highly specific to the domain of semantic adaptation. We believe that the DSML and the execution semantics are, as a principle, expressive enough to model different semantic adaptations, though additional constructs might be needed for additional behaviours. Furthermore, a model-to-text transformation is defined to generate the adaptation block code for the *ModHel’X* framework. The DSML enables the modeller to easily customise adaptation blocks that fit the heterogeneous models in a modular way, as the combined models are left untouched. We were able to reconstruct the default Ptolemy II [56] semantic adaptation of Discrete Event models to Synchronous Data Flow models, and corrected it with our own model. The DSML models are only tens of lines of code, and are transformed to hundreds of lines of Java code. Both DSMLs are mapped to the *ModHel’X* Java framework. The visual syntax of the DSML in *AToM^3* for heterogeneous modelling reflects the graph-like nature of the models. Moreover, dozens of elements are translated to hundreds of lines of Java code. Similarly, the adaptation block models are only tens of lines of code, and are transformed to hundreds of lines of Java code.

Our work on the composition of languages presents solutions in a specific context: we employed *metaDepth*'s extension mechanisms on the one hand and *ModHel'X*'s heterogeneous modelling framework on the other hand. In contrast to previous chapters, these contributions stem from bottom-up needs in existing frameworks. We refrained from presenting an overall view on the broad topic of composition. In short, this chapter describes research that is more anecdotal than that in the previous chapters.

The work on the syntactic composition of modelling languages in *metaDepth* was the result of a collaboration with Juan de Lara and Esther Guerra (Universidad Autónoma de Madrid), who built *metaDepth*. I implemented the approach with the support of Juan de Lara. The work on heterogeneous modelling and semantic adaptation in *ModHel'X* was joint research with Joachim Denil (University of Antwerp). We worked together with Frédéric Boulanger, Christophe Jacquet and Cécile Hardebolle (École Supérieure d'Électricité), the builders of *ModHel'X*, and Hans Vangheluwe. Joachim Denil and I elaborated the approach and I implemented the DSMLs in *AToM*^{3} and *metaDepth*.

5.2 Future Work

The presented research led to the identification of important threads for future work in the domain of Multi-Paradigm modelling. The following is future work that I believe can have a significant impact.

5.2.1 Modelling of Semantics

In current DSM approaches, the semantic mapping is modelled as a transformation. From the research presented in this thesis, it becomes clear that this often does not suffice. We presented an approach for modelling properties, which can be considered models of requirements that encompass a significant part of the system's semantics, describing its intent. Our contribution in property modelling is however only the first step to achieving this. Relationships could be defined on these properties, so that it becomes possible to reason about abstraction and refinement. In the context of language evolution, this can be an important tool to ensure language continuity. In this thesis, evolution is modelled as a delta model. Properties could be used to capture the intent of a language evolution. This additional information might further automate the consequences of evolution. Recent work by Barroca *et al.* takes on this challenge by combining ontology engineering with MPM [12]. In their work, both properties and models are explicitly modelled. An ontology is represented by a set of properties and their relations. An ontological conformance relationship can be defined between models and ontologies. Consequently, relationships between properties result in relationships between models, enabling one to reason about consistency, abstraction and refinement.

In the context of simulation, the semantics of a language can be captured in a model of computation. In simulation, approximations are made to result in a computable (*i.e.,* discrete-time) execution. Hence, this execution can be modelled in terms of execution steps [61] as (a variant of) a timed automaton. Additionally, we believe that many of the concepts we presented in this thesis for semantic adaptation can be reused, such as time calculus, expressions about pin values, etc. Using automata as a model of computation provides the means to instrument the model of computation. In the context of language composition, this allows fine-grained composition of semantics by weaving different models of computation. In the context of semantic adaptation, constraints on adaptation blocks can be defined to make sure that it models a correct semantic adaptation between the given models of computation. Other potential benefits are better modularity, opportunities for reuse, analysis, platform independence, inter-tool operability and other benefits of MPM. Moreover, Van Mierlo *et al.* aim for explicitly modelling of debugging using a similar approach [139]. Note that this assumes a white box approach, as opposed to the approach presented in this thesis.

5.2.2 Tool Support for Multi-Paradigm Modelling

Metamodelling and model transformations are the enablers for language engineering in Domain-Specific Modelling. Many tools have been developed that support metamodelling and model transformation, including the tools used in this thesis, *AToMPM*, *metaDepth* and *AToM^3*. To maximally benefit from MPM, mature tools are necessary that adhere to MPM principles. To our knowledge, there are currently no tools that offer a fully modelled linguistic metalanguage, including the expressiveness of an action language and inter-model relationships. A model repository called the Modelverse is currently under development by our research group MSDL to support this [196].

An essential part of modelling tools is how models are represented and manipulated. In the context of language engineering, this brings us to concrete syntax modelling. In terms of visual syntax, approaches are still at their infancy [143]. No de facto approach such as metamodelling exists for modelling concrete syntaxes. Additionally, DSM approaches mainly focus on visualisation. For model manipulation, standard modelling environments are used. MPM's credo "using the most appropriate formalism" should be extended to the development of the most appropriate modelling environments. These modelling environments should themselves be modelled according to MPM principles. Since the environment's behaviour is reactive and contains complicated objects, a combination of Statecharts and Class Diagrams is a promising basis from modelling such environments. Concrete syntax modelling, both for representation and manipulation, can follow a "default first" approach. Often, quick prototyping is desired, and the language engineer does not want to get lost in extensive modelling of concrete syntax and modelling environment. In that case, default concrete syntax (*e.g.,* object diagrams) and default modelling environments (*e.g., AToM^3*'s or *AToMPM*'s generated DSML environments) can be used so that no effort at all is required for concrete syntax modelling. The other extreme is a fully customised modelling environment, with custom icons, different views on the model, custom reaction to keystrokes, custom layout algorithms, etc. Such an environment could be useful for *e.g.,* the metamodelling language, potentially supporting the same user experience as hard-coded UML tools. The benefits over hard-coded UML tools are manifold: analysis of user interaction, custom debugging, fine-grained undo and redo operations, customisation and extension of the environment, etc.

5.2.3 Standardisation and Availability of Model Repositories

Once new approaches for MPM have been researched, there is a need for usability studies and empirical research in order to validate research. To carry out usability studies, adequate tool support of the framework under study is required, as described above, in order to avoid that the results of the usability studies are polluted by accidental complexity that comes with a malfunctioning tool.

Empirical research requires a collection of available models. In contrast to research in code-base software engineering, there is a lack of widely available model repositories. The ReMoDD initiative by France *et al.* tries to address this issue [64] and aim for a generally accessible model repository. For interoperability, a standard model format needs to be proposed. Currently, the Ecore metamodel of the Eclipse Modeling Framework (EMF) [188] is the most widely used format. This does not include a standard for metamodelling (including constraints and actions), concrete syntax modelling and model transformation.

Additionally, in order to share their models, companies need the ability to hide sensitive data. Subsequently models need to be sanitised or obfuscated, so that the model can be only queried in a controlled way. An implementation can be found in the VIATRA transformation tool [39].

5.2.4 Industrial Adoption

DSM has been successfully used in industry, notably by use of MetaCase's DSM tool MetaEdit+ [96]. The topics of this thesis however, are contributions to the foundations of DSM, and more specifically MPM. We supported our findings with implementation prototypes, but our contributions do not have the maturity for industrial adoption. An important factor for industrial adoption is a usable tool. This implies that usability studies have to be conducted, as discussed above. Moreover, in the context of our contributions, an adequate MPM tool is a prerequisite for a user-friendly implementation of our contributions. Of all contributions in this thesis, the *ProMoBox* framework of Chapter 2 is the most advanced candidate for a usability study. A second factor, related to usability, is the simplification of our contributions so that solutions to the "most common" problems are supported. The goal is to lower the bar for learning the solutions. A third factor is empirical research to further validate the claims of this thesis in an industrial context and to assess what the "most common" problems are.

APPENDIX A
Manually Created Elevator Model

(#floors, #buttons)	original [134]				adapted (Figure A.1)			
	states	trans.	time (s)	mem. (KB)	states	trans.	time (s)	mem. (KB)
(4, 4)	3145	17948	0.021	240	5989	13765	0.032	594
(5, 5)	16456	110261	0.139	1444	15362	40617	0.096	1699
(6, 6)	81451	620053	0.847	7768	37423	111737	0.266	4568

Table A.1: Comparison of the results of the adapted manual Promela model and the original model by [134].

The algorithm of Listing A.1 is based on an exemplary Promela model from Merz and Navet's "on the verification of real-time systems" [134]. The original model consists of four floors and four buttons, each requesting a floor. The elevator controller and each of the four buttons each run in a parallel process. The different possibilities of the controller are options in a do loop, including moving up or down and opening the door (which is our interpretation of `moving == false`), or changing directions in case no further can be moved or if a request is in the elevator's path. Every step is atomic, which means that a button cannot be pressed during the execution of a step.

Our adaptation has the following differences to make it semantically equivalent to our DSML:

— the concept of UpButtons and DownButtons is introduced, for which the elevator will not stop when it is traveling in the opposite direction;

— rules are split to correspond to the rules of the operational semantics of Figure 1.7, and so that conceptual steps (moment of input and output) correspond;

— the application of the environment (*i.e.,* pressing a button) is more restricted and can only happen after a state change, according to the annotated operational semantics. To implement this, an inner do loop is introduced to apply rules as long as no conceptual step has finished;

— the variables `tried1` and `tried2` are introduced to denote rules that have been tried. This is necessary because many rules have conditions that are spread over multiple conditions. Without these extra variables, an infinite loop occurs if a rule is partly, but not fully, matched;

— after finishing a conceptual step (*i.e.,* `step_taken` becomes true), at most one button can be pressed.

As a result of these changes, the adaptation yields slightly lower performance in case of four floors, and a better performance than the original model for larger numbers of floors as shown in Table A.1.

```
#define FLOORS 3
int floor = 0;
bool doors_open = false; bool going_up = true;

typedef Floor {
  bool button = false;
  bool up = false;
  bool down = false;
}

bool step_taken = false

Floor f[FLOORS];

proctype lift() {
  int j, k;
  bool tried1, tried2;
  do ::
    // tried variables, to avoid infinite loop trying to match
    // one option that has an inner condition
    tried1 = false;
    tried2 = false;
    do
    :: !step_taken ->
      if
      // open door
      :: !doors_open && going_up && f[floor].button -> atomic {
         doors_open = true; f[floor].button = false; step_taken = true     }
      :: !doors_open && going_up && f[floor].up -> atomic {
         doors_open = true; f[floor].up = false; step_taken = true     }
      :: !doors_open && !going_up && f[floor].button -> atomic {
         doors_open = true; f[floor].button = false; step_taken = true     }
      :: !doors_open && !going_up && f[floor].down -> atomic {
         doors_open = true; f[floor].down = false; step_taken = true     }
      // close door
      :: !tried1 && doors_open -> atomic {
         tried1 = true;
         j = 0;
         do
         :: j == FLOORS -> break
         :: j < FLOORS && (f[j].button || f[j].up || f[j].down) ->
             // there are other requests
             doors_open = false; step_taken = true; break
         :: else -> j++
         od     }
      // move
      :: !tried1 && !doors_open && going_up && !f[floor].button && !f[floor].up -> atomic {
         tried1 = true;
         j = floor+1;
         do
         :: j == FLOORS -> break
         :: j < FLOORS && (f[j].button || f[j].up || f[j].down) ->
             // request in the elevator's path
             floor++; step_taken = true; break
         :: else -> j++
         od     }
      :: !tried1 && !doors_open && !going_up && !f[floor].button && !f[floor].down -> atomic {
         tried1 = true;
         j = floor-1;
         do
         :: j < 0 -> break
         :: j >= 0 && (f[j].button || f[j].up || f[j].down) ->
             // request in the elevator's path
             floor--; step_taken = true; break
         :: else -> j--
```

```
          od      }
        // change directions
        :: !tried2 && going_up && !f[floor].button && !f[floor].up -> atomic {
          tried2 = true;
          j = floor+1;
          do
          :: j == FLOORS -> // no requests in the elevator's path
              k = floor-1;
              do
              :: k < 0 -> j++; break
              :: k >= 0 && (f[k].button || f[k].up || f[k].down) ->
                 // requests in the reversed path, change direction
                 going_up = false; j++; tried1 = false; tried2 = false; break
              :: else -> k—
              od
          // still requests in the elevator's path, do not change direction
          :: j < FLOORS && (f[j].button || f[j].up || f[j].down) -> break
          // if successfully changed direction, break out of this loop
          :: j > FLOORS -> break
          :: else -> j++
          od      }
        :: !tried2 && !going_up && !f[floor].button && !f[floor].down -> atomic {
          tried2 = true;
          j = floor-1;
          do
          :: j == -1 -> // no requests in the elevator's path
             k = floor+1;
             do
             :: k == FLOORS -> j—; break
             :: k < FLOORS && (f[k].button || f[k].up || f[k].down) ->
                // requests in the reversed path, change direction
                going_up = true; j—; tried1 = false; tried2 = false; break
             :: else -> k++
             od
          // still requests in the elevator's path, do not change direction
          :: j >= 0 && (f[j].button || f[j].up || f[j].down) -> break
          // if successfully changed direction, break out of this loop
          :: j < -1 -> break
          :: else -> j—
          od      }
        :: else -> break // if there are no requests
        fi;
      :: else -> break // step was taken
      od;
      step_taken = true;
      if
      :: !step_taken -> skip
      fi
  od
}

proctype button_(int myfloor) {
  do :: atomic { step_taken ->
    if
      :: true -> skip
      :: f[myfloor].button == false -> f[myfloor].button = true
    fi;
    step_taken = false; }
  od
}

proctype upbutton_(int myfloor) {
  do :: atomic { step_taken ->
    if
      :: true -> skip
      :: f[myfloor].up == false -> f[myfloor].up = true
    fi;
    step_taken = false; }
  od
}

proctype downbutton_(int myfloor) {
  do :: atomic { step_taken ->
    if
      :: true -> skip
      :: f[myfloor].down == false -> f[myfloor].down = true
    fi;
    step_taken = false; }
  od
}

init {
  run lift();
  int i = 0; do :: i < FLOORS -> run button_(i); i++ :: else -> break od;
  i = 0; do :: i < FLOORS-1 -> run upbutton_(i); i++ :: else -> break od;
  i = 1; do :: i < FLOORS -> run downbutton_(i); i++ :: else -> break od;
}

ltl reachesFloor {
[] (! (f[0].button || f[0].up)
   || <> ((f[0].button || f[0].up) && <> (doors_open && floor == 0))) &&
[] (! (f[1].button || f[1].down || f[1].up)
   || <> ((f[1].button || f[1].down || f[1].up)
         && <> (doors_open && floor == 1))) &&
[] (! (f[2].button || f[2].down)
   || <> ((f[2].button || f[2].down) && <> (doors_open && floor == 2))) }
```

Listing A.1: .]A manually created elevator model, based on [134].

APPENDIX B
Delta Operations

This appendix lists the definitions of all delta operations of Table 3.1. Similar to Herrmannsdörfer's list of metamodel changes [86], more operations can be added by creating a new definition.

Generalise property

Description: Generalise the multiplicity of a property (attribute or association), allowing more instances.
Type: additive
Impact: non-breaking
Migration operation: none
Inverse operation: Restrict property
Function signature: `GeneraliseProperty(prop, lowsrc, highsrc[, lowdst, highdst])`
Parameters:

— `prop` (property): the property to be generalised

— `lowsrc` (integer): the new lower bound at the source

— `highsrc` (integer): the new upper bound at the source

— `lowdst` (integer): (in case of an association) the new lower bound at the target

— `highdst` (integer): (in case of an association) the new upper bound at the target

Preconditions:

— `highsrc` > 0, `highdst` > 0

— all new bounds are the same or more general

Add class

Description: Add a new class, determine whether it is abstract and determine the superclass. In case of an abstract class, this change is in fact updative.
Type: additive
Impact: non-breaking
Migration operation: none
Inverse operation: Eliminate class
Function signature: `AddClass(name, superclass, abstract)`
Parameters:

— `name` (string): the name of the new class

— `superclass` (class): the superclass of the new class, or none

— `abstract` (boolean): whether the new class is abstract

Preconditions:

— `name` is not yet taken

Add non-obligatory property

Description: Add a new property (attribute or association), with non-restrictive lower multiplications of 0.
Type: additive
Impact: non-breaking
Migration operation: none
Inverse operation: Eliminate property
Function signature: `AddNonObligatoryProperty(name, owner, highsrc, type[, highdst][, default])`
Parameters:

- `name` (string): name of the new property
- `owner` (class): the owner of the property
- `highsrc` (integer): the upper bound at the source
- `type` (type or class): the target (in case of an association) or type (in case of an attribute) of the property
- `highdst` (integer): (in case of an association) the upper bound at the target
- `default` (any): (in case of an attribute, optional) the default value

Preconditions:

- `highsrc` > 0, `highdst` > 0
- `default` is of type `type`, or none
- `name` is not yet taken by another of `owner`'s properties

Make class concrete

Description: Make an abstract class non-abstract.
Type: additive
Impact: non-breaking
Migration operation: none
Inverse operation: Make class abstract
Function signature: `MakeClassConcrete(class)`
Parameters:

- `class` (class): the class to be made non-abstract

Preconditions:

- `class` is abstract

Extract abstract superclass

Description: Extract an abstract superclass from given subclasses.
Type: updative
Impact: non-breaking

Migration operation: none
Inverse operation: Flatten abstract hierarchy
Function signature: `ExtractAbstractSuperclass(name, subclasses)`
Parameters:

- `name` (string): the name of the new class
- `subclasses` (set<class>): the subclasses of the new class

Preconditions:

- no inheritance cycles can be introduced

Extract superclass

Description: Extract a non-abstract superclass from given subclasses.
Type: additive
Impact: non-breaking
Migration operation: none
Inverse operation: Flatten hierarchy
Function signature: `ExtractSuperclass(name, subclasses)`
Parameters:

- `name` (string): the name of the new class
- `subclasses` (set<class>): the subclasses of the new class

Preconditions:

- no inheritance cycles can be introduced

Flatten abstract hierarchy

Description: Remove an abstract class, propagating all properties and inheritance links to its subclasses, while maintaining inheritance links between subclasses and superclass.
Type: updative
Impact: non-breaking
Migration operation: none
Inverse operation: Extract abstract superclass
Function signature: `FlattenAbstractHierarchy(class)`
Parameters:

- `class` (class): the subclasses of the new class

Preconditions:

- `class` is abstract

Push property from abstract class

Description: Move a property in an abstract class to all of its owner's subclasses.
Type: updative
Impact: non-breaking
Migration operation: none

Inverse operation: Pull property to abstract class
Function signature: `PushPropertyFromAbstractClass(prop)`
Parameters:

— `prop` (property): the property that will be moved to the subclasses

Preconditions:

— `prop`'s owner has at least one subclass

— `prop`'s owner is abstract

Pull property to abstract class

Description: Move a common property to its owner's abstract superclass that has no concrete (indirect) subclasses other than the property's owner.
Type: updative
Impact: non-breaking
Migration operation: none
Inverse operation: Push property from abstract class
Function signature: `PullPropertyToAbstractClass(prop)`
Parameters:

— `prop` (property): the property that will be moved to the superclass

Preconditions:

— `prop`'s owner has a superclass

— `prop`'s owner is abstract

— `prop`'s owner does not transitively have non-abstract subclasses other than `prop`'s owner

— `prop` is present and the same in all subclasses of `prop`'s owner's superclass

Eliminate class

Description: Remove a class, and its properties, while maintaining inheritance links between subclasses and superclass.
Type: subtractive
Impact: breaking and resolvable
Migration operation: eliminate instances
Inverse operation: Add class and Add non-obligatory property
Function signature: `EliminateClass(class)`
Parameters:

— `class` (class): the class to be removed

Preconditions:

— none

Eliminate property

Description: Remove a property.
Type: subtractive
Impact: breaking and resolvable
Migration operation: eliminate property instances
Inverse operation: Add (non-)obligatory property
Function signature: `EliminateProperty(prop)`
Parameters:

— `prop` (property): the property that will be removed

Preconditions:

— none

Make class abstract

Description: Make a non-abstract class abstract.
Type: subtractive
Impact: breaking and resolvable
Migration operation: eliminate class instances
Inverse operation: Make class concrete
Function signature: `MakeClassAbstract(class)`
Parameters:

— `class` (class): the class to be made abstract

Preconditions:

— `class` is not abstract

Extract class

Description: Extract a class and a number of properties to a new class, and add an association with multiplicity 1 from the old class to the new class.
Type: updative
Impact: breaking and resolvable
Migration operation: for each instance, create additional instance and move properties
Inverse operation: Inline class
Function signature: `ExtractClass(features, name, associationname)`
Parameters:

— `features` (set<feature>): features to be extracted

— `name` (string): name of the new class

— `associationname` (string): name of the new association

Preconditions:

— `features` are all of the same class

Inline class

Description: Inline a class and its properties to a class to which it is associated with a one-to-one mapping, and remove the association with multiplicity 1.
Type: updative
Impact: breaking and resolvable
Migration operation: for each instance, remove properties and instance
Inverse operation: Inline class
Function signature: `ExtractClass(features, name, associationname)`
Parameters:

— `class` (class): the to be inlined class

— `property` (property): the to be inlined property

Preconditions:

— `property` has multiplicity 1 on both sides

Flatten Hierarchy

Description: Remove a non-abstract class, propagating all properties and inheritance links to its subclasses, while maintaining inheritance links between subclasses and superclass.
Type: subtractive
Impact: breaking and resolvable
Migration operation: eliminate superclass instances
Inverse operation: Extract superclass
Function signature: `FlattenHierarchy(class)`
Parameters:

— `class` (class): the subclasses of the new class

Preconditions:

— `class` is not abstract

Push property

Description: Move a property in a non-abstract class to all subclasses.
Type: subtractive
Impact: breaking and resolvable
Migration operation: eliminate properties from superclass instances
Inverse operation: Pull property
Function signature: `PushProperty(prop)`
Parameters:

— `prop` (property): the property that will be moved to the subclasses

Preconditions:

— `prop`'s owner has at least one subclass

— `prop`'s owner is not abstract

Rename class

Description: Rename a given class.
Type: updative
Impact: breaking and resolvable
Migration operation: change type of instances
Inverse operation: Rename class
Function signature: `RenameClass(class, name)`
Parameters:

— `class` (class): the class to be renamed

— `name` (string): the new name of the class

Preconditions:

— `name` is not yet taken

Rename property

Description: Rename a given property.
Type: updative
Impact: breaking and resolvable
Migration operation: change type of property instances
Inverse operation: Rename property
Function signature: `RenameProperty(prop, name)`
Parameters:

— `prop` (property): the property to be renamed

— `name` (string): the new name of the property

Preconditions:

— `name` is not yet taken by another of `prop`'s owner's properties

Add obligatory property

Description: Add a new property (attribute or association) with lower multiplicity of at least 1
Type: additive
Impact: breaking and unresolvable
Migration operation: add property instances with default values
Inverse operation: Eliminate property
Function signature: `AddObligatoryProperty(name, owner, lowsrc, highsrc, type[, lowdst, highdst][, default])`
Parameters:

— `name` (string): name of the new property

— `owner` (class): the owner of the property

— `lowsrc` (integer): the lower bound at the source

— `highsrc` (integer): the upper bound at the source

— `type` (type or class): the target (in case of an association) or type (in case of an attribute) of the property

— `lowdst` (integer): (in case of an association) the lower bound at the target

— `highdst` (integer): (in case of an association) the upper bound at the target

— `default` (any): (in case of an attribute, optional) the default value

Preconditions:

— `highsrc` > 0, `highdst` > 0, `lowsrc` $\leq$ `highsrc`, `lowdst` $\leq$ `highdst`, `lowsrc` > 0 or `lowdst` > 0

— `default` is of type `type`, or none

— `name` is not yet taken by another of `owner`'s properties

Pull property

Description: Move a common property to the superclass.
Type: additive (or updative, in the case where the superclass is abstract and it has no concrete (indirect) subclasses other than the property's owner)
Impact: breaking and unresolvable
Migration operation: add property instances with default values for superclass instances
Inverse operation: Push property
Function signature: `PullProperty(prop)`
Parameters:

— `prop` (property): the property that will be moved to the superclass

Preconditions:

— `prop`'s owner has a superclass

— `prop` is present and the same in all subclasses of `prop`'s owner's superclass

Restrict property

Description: Restrict the multiplicity of a property (attribute or association), allowing less instances.
Type: subtractive
Impact: breaking and unresolvable
Migration operation: remove property instance if non-compliant
Inverse operation: Generalise property
Function signature: `RestrictProperty(prop, lowsrc, highsrc[, lowdst, highdst])`
Parameters:

— `prop` (property): the property to be generalised

— `lowsrc` (integer): the new lower bound at the source

— `highsrc` (integer): the new upper bound at the source

— `lowdst` (integer): (in case of an association) the new lower bound at the target

— `highdst` (integer): (in case of an association) the new upper bound at the target

Preconditions:

— `highsrc` > 0, `highdst` > 0, `lowsrc` $\leq$ `highsrc`, `lowdst` $\leq$ `highdst`

— all new bounds are the same or more restricted

APPENDIX C
Prototype of Language Evolution in *metaDepth*

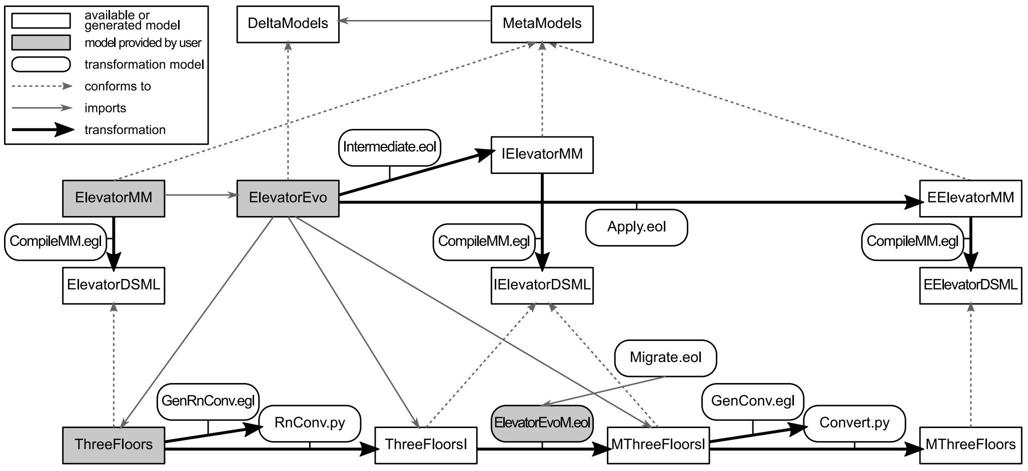


Figure C.1: Evolution of the *Elevator* language and co-evolution of instance models with *metaDepth*.

In this appendix the approach of Section 3.3 is illustrated with a prototype of the *Elevator* evolution scenario. First, a detailed implementation for language evolution and model co-evolution is presented. Then, an example of co-evolution of the operational semantics transformation is presented.

A prerequisite to the approach is metamodel transformation. Unfortunately, this is only limitedly supported by *metaDepth*, as there is no explicit metamodel of metamodels. To this end we created a metamodelling language called MetaModels with a code generator to *metaDepth* metamodels. This way, MetaModels instances can be subjected to transformations. Additionally *metaDepth* metamodels can be generated to maintain the metamodelling support (inheritance, conformance, links, attributes, etc.). This can be considered an implementation of the operationalise relationship presented in Section 1.2.4.

The following is assumed in this implementation:

— the difference model is available;

— for simplicity, only one intermediate metamodel is generated instead of one for each operation;

— for simplicity, the migration pipeline is not optimised, meaning that the checkout transformations γ_i are still evaluated;

— transformations can be executed on models conform to the intermediate metamodel. This means that conversion steps from and to the intermediate model are not needed in this implementation.

The *metaDepth* implementation of language evolution and model co-evolution is summarised in Figure C.1. The step-by-step approach produces following models:

1. ElevatorMM (see Listing C.2): a metamodel is created, as an instance of the MetaModels language (see Listing C.1). This is a simple metalanguage modelling classes with inheritance, properties (attributes or associations) and a number of constraints in EOL. The metamodel is an explicit element and serves as a container of all metamodel elements which will be useful

in transformations for iterating over all elements[1]. In the instance, the container can be left empty so that nu redundant information needs to be modelled. If left empty, all elements will be automatically set by a transformation that relies on this attribute.

2. ElevatorDSML (see Listing C.4): from ElevatorMM a *metaDepth* metamodel ElevatorDSML will be generated as explained above, by the CompileMM EGL script (see Listing C.3). The script finds all classes and associations either in the elements container, or by finding all class and association instances. The names of classes are extended with "Type", to avoid name clashes with ElevatorMM elements. The resulting model can be loaded as a metamodel in *metaDepth* and is the *metaDepth* equivalent of the *Elevator* metamodel presented in Figure 1.4.

3. ThreeFloors (see Listing C.5): an instance of ElevatorDSML is modelled. This is the *metaDepth* equivalent of Figure 1.6. This model imports the delta model called ElevatorEvo, as this will be necessary in later steps.

4. ElevatorEvo (see Listing C.7): the delta model ElevatorEvo is a model containing the delta operations of Table 3.3. It is an instance of DeltaModels (see Listing C.6), a DSML for difference modelling, containing all operations as presented in Table 3.1, and more detailed in Appendix B. For each operation, the parameters are attributes and the attribute constraints (only shown for one operation) are modelled as EOL constraints. Names of classes or properties are strings, because it might be possible that, when loading the difference model in *metaDepth*, referenced classes cannot be found yet as they are created or renamed in an earlier delta step. This is the case for CallButton. The delta model imports the metamodel ElevatorMM, which is necessary for subsequent steps.

5. EElevatorDSML (see Listing C.10): the evolved metamodel ElevatorDSML is generated by first applying the delta model ElevatorEvo to the original metamodel ElevatorMM and then compiling the metamodel with CompileMM. Applying the delta model is done by the Apply transformation (see Listing C.8), which implements a generic `apply` operation for every delta operation. The Apply transformation relies on helper operations defined in (see Listing C.9 – not shown in Figure C.1), which includes operations on the metalevel, instance level and concerning instantiation. Note that because of a technical issue, not all links of a certain association with n-to-n multiplicity might be retrieved. In the *Elevator* example this issue does not apply, because there is no such association.

6. IElevatorDSML (see Listing C.12): in a similar way, the intermediate metamodel IElevatorDSML is generated by first merging the delta model ElevatorEvo to the original metamodel ElevatorMM and then compiling the metamodel with CompileMM. The Intermediate transformation (see Listing C.11) implements the merge by again providing a generic operation for each delta operation. For some operations (*e.g.,* EliminateClass) nothing needs to be changed to obtain the intermediate metamodel. The output trace of executing Intermediate is shown in Listing C.13. This concludes the operation on the language level. What remains is the actual migration of instances.

[1]Usually, all instances of a certain node in the scope of a model can be iterated, but it turns out that after transformation, metamodel elements might be scattered across different models. To this end, the elements container serves as the correct reference for all elements of the metamodel in question.

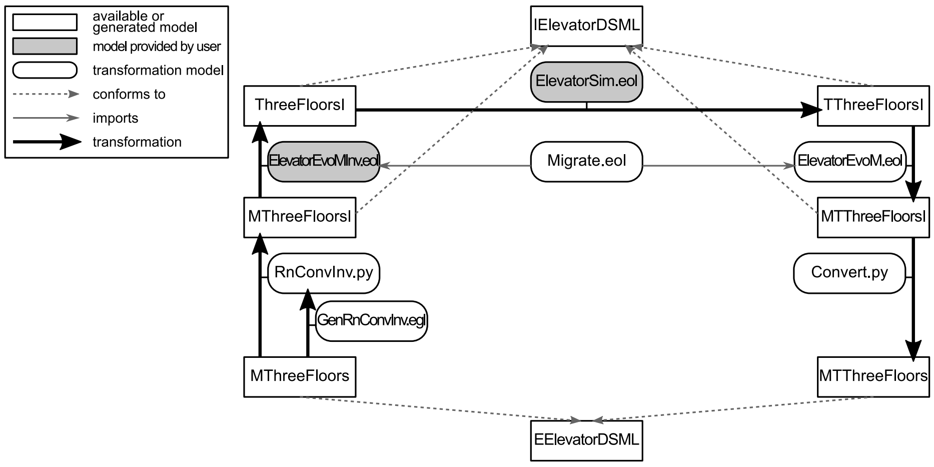


Figure C.2: Co-evolution of the operational semantics transformation with *metaDepth*.

7. ThreeFloorsI (see Listing C.14): the ThreeFloors instance model is converted to the intermediate metamodel to become a virtual version 0 instance. In this process, all rename operations δ_r are applied. The conversion is done outside of *metaDepth* by a generated search-and-replace Python script. This is done with a script because it is not possible to cast an instance to a different metamodel in *metaDepth*, and a full copy transformation is cumbersome and not straightforward for such a simple task. Note that this rename issue is well-known in co-evolution [77]. Apart from renaming, the model is still exactly the same as ThreeFloors.

8. MThreeFloorsI: the virtual version 0 instance ThreeFloorsI is migrated to the virtual version 1 instance MThreeFloorsI by applying the customised ElevatorEvoM migration transformation (see Listing C.16). This transformation starts from the default migration operations, which are implemented in the Migrate transformation (see Listing C.15). In *metaDepth*, the migration operations are not generated from the delta model. Instead, a migration operation is defined for every delta operation, but is applied to the instance model. Non-breaking operations have an empty operation body. ElevatorEvoM starts from these default migration operations and allows the language engineer to define customised migration operations, such as replacing UpButtons and DownButtons with FloorButtons with the right up_direction value. They are inserted in the migration pipeline. The output trace of execution ElevatorEvoM is shown in Listing C.17.

9. MThreeFloors (see Listing C.18): the virtual version 1 instance MThreeFloorsI is retyped with a generated Python script to produce the migrated MThreeFloors instance.

The *metaDepth* implementation of transformation co-evolution is summarised in Figure C.2. The transformation implementing the *Elevator* operational semantics is migrated. Note that this is an endogenous transformation, which means that after evolution of the *Elevator* DSML, the transformation is both subject to domain evolution and image evolution. As explained in Section 3.3, a number of artifacts produced by the model migration phase can be reused. The step-by-approach uses or produces the following models:

1. MThreeFloors (see Listing C.18): we start from the migrated instance of the model migration phase, and retyped and inverse rename operations δ_r^{-1} are applied by a generated Python script to produce MThreeFloorsI, the same virtual version 1 instance as in the model migration phase;

2. ElevatorEvoInv (see Listing C.19): starts from the default migration operations of the inverse delta operations and allows the language engineer to define customised migration operations, such as replacing FloorButtons with UpButtons and DownButtons. The result of this transformation is ThreeFloorsI, the same virtual version 0 instance as in the model migration phase;

3. ElevatorSim (see Listing C.20): the original operational semantics transformation as presented in Section 1.3, which produces the virtual version 0 instance TThreeFloorsI. The output trace of executing ElevatorSim is shown in Listing C.21.

4. MTThreeFloors (see Listing C.22): TThreeFloorsI is migrated to the virtual version 1 artifact MTThreeFloorsI, and is converted using the generated Python script from the model migration phase to produce the migrated MTThreeFloors instance.

Note that in this case, the transformation model was not edited. In order to incorporate EmergencyButtons in the operational semantics, it needs to be edited. This would however require the operational semantics to include an environment, as discussed in Section 2, so that buttons can be pressed at run-time.

```
Model MetaModels {
  enum PrimitiveType { BOOLEAN, INT, DOUBLE, STRING }
  Node MetaModel[1] { elements : MetaElement[*]; }
  abstract Node MetaElement {}
  Node MetaClass : MetaElement {
    is_abstract : boolean = false;
    superclass : MetaClass[0..1];
    properties : MetaProperty[*]; }
  abstract Node MetaProperty : MetaElement {
    lowsrc : int = 0;
    highsrc : int = -1;
    lowsrcValid : $self.lowsrc >= 0 and (self.highsrc == -1 or self.lowsrc <= self.highsrc)$
    highsrcValid : $self.highsrc > 0 or self.highsrc == -1$ }
  Node MetaAttribute : MetaProperty {
    type : PrimitiveType;
    lowsrc : int = 1;
    highsrc : int = 1;
    default : String = ""; } // default value is always a string; its value will be parsed depending on the type
  Node MetaAssociation : MetaProperty {
    target : MetaClass;
    lowdst : int = 0;
    highdst : int = -1;
    lowdstValid : $self.lowdst >= 0 and (self.highdst == -1 or self.lowdst <= self.highdst)$
    highdstValid : $self.highdst > 0 or self.highdst == -1$ }
  noInheritanceCycles : $MetaClass.allInstances()
                          .includingAll(MetaModel.allInstances().first().elements
                                          .select(e | e.isTypeOf(MetaClass)))
                          .asSet().forAll(n1 | not MetaClass.allInstances()
                            .includingAll(MetaModel.allInstances().first().elements
                                            .select(e| e.isTypeOf(MetaClass)))
                            .asSet().select(n2 | n2 == n1)
                            .closure(parent | parent.superclass).includes(self))$
  presentInNode : $MetaProperty.allInstances()
                   .includingAll(MetaModel.allInstances().first().elements
                                   .select(e | e.isTypeOf(MetaProperty)))
                   .asSet().forAll(p | MetaClass.allInstances()
                     .includingAll(MetaModel.allInstances().first().elements
                                     .select(e | e.isTypeOf(MetaClass)))
                     .asSet().select(n | n.properties.isDefined()
                                     and n.properties.includes(p)).size() == 1)$
}
```

Listing C.1: `MetaModels.mdepth`: the model of the metalanguage.

```
load "MetaModels"
MetaModels ElevatorMM {
  MetaModel ElevatorDSML { elements = []; }
  MetaClass Elevator { properties = [doors_open, going_up, currentfloor, elevator_button]; }
  MetaClass Floor { properties = [nr, next]; }
  MetaClass Button { is_abstract = true; properties = [pressed, requests]; }
  MetaClass ElevatorButton { superclass = Button; }
  MetaClass FloorButton { superclass = Button; is_abstract = true; }
  MetaClass UpButton { superclass = FloorButton; }
  MetaClass DownButton { superclass = FloorButton; }
  MetaAttribute nr { type = INT; }
  MetaAttribute doors_open { type = BOOLEAN; }
  MetaAttribute going_up { type = BOOLEAN; }
  MetaAttribute pressed { type = BOOLEAN; }
  MetaAssociation currentfloor { target = Floor; lowsrc = 1; highsrc = 1; lowdst = 0; highdst = -1;}
  MetaAssociation next { target = Floor; lowsrc = 0; highsrc = 1; lowdst = 0; highdst = 1; }
  MetaAssociation requests { target = Floor; lowsrc = 1; highsrc = 1; lowdst = 0; highdst = -1; }
  MetaAssociation elevator_button { target = ElevatorButton; lowsrc = 0; highsrc = -1; lowdst = 1; highdst = 1; }
}
```

Listing C.2: `ElevatorMM.mdepth`: the metamodel of *Elevator* as an instance of the metalanguage.

```
// generated
[%
var MetaClasses : Bag(MetaClass);
if (MetaModel.allInstances().first().elements.size() > 0) {
  MetaClasses = MetaModel.allInstances().first().elements.select(e | e.isTypeOf(MetaClass));
} else {
  MetaClasses = MetaClass.allInstances().asBag();
}
var MetaAssociations : Bag(MetaAssociation);
if (MetaModel.allInstances().first().elements.size() > 0) {
  MetaAssociations = MetaModel.allInstances().first().elements.select(e | e.isTypeOf(MetaAssociation));
} else {
  MetaAssociations = MetaAssociation.allInstances().asBag();
}
%]
Model [%= MetaModel.allInstances().first() %] {
[% var compiled : Sequence(MetaClass); %]
[% while (MetaClasses.size() <> compiled.size()) { %]
  [% var n : MetaClass = MetaClasses.select(n | not compiled.includes(n) and (not n.superclass.isDefined() or compiled.includes(n.superclass))).first(); %]
  [% compiled.add(n); %]
  [% if (n.is_abstract) { %]  abstract Node[% } else { %]  Node[% } %] [%= n %]Type[% if (n.superclass.isDefined()) { %] : [%= n.superclass %]Type[% } %] {
  [% for (a in n.properties.select( f | f.isTypeOf(MetaAttribute))) { %]
    [%= a %] : [% if (a.type.toString() == "STRING") { %]String[% } else {%][%= a.type.toString().toLowerCase() %][% } %][% if (not (a.lowsrc == 1 and a.highsrc == 1)) { %][[%= a.lowsrc %]..[% if (a.highsrc == -1) { %]*[% } else { %][%= a.highsrc %][% } %]][% } %][% if (a.default.isDefined()) { %] = [%= a.default %][% } %];
  [% } %]
  [% for (a in n.properties.select( f | f.isTypeOf(MetaAssociation))) { %]
    [%= a %]_outgoing : [%= a.target %]Type[[%= a.lowsrc %]..[% if (a.highsrc == -1) { %]*[% } else { %][%= a.highsrc %][% } %]];
  [% } %]
  [% for (src in MetaClasses.select(src | src.properties.select(f | f.isTypeOf(MetaAssociation) and f.target == n).size() > 0)) { %]
  [% for (a in src.properties.select( f | f.isTypeOf(MetaAssociation) and f.target == n)) { %]
    [%= a %]_incoming : [%= src %]Type[[%= a.lowdst %]..[% if (a.highdst == -1) { %]*[% } else { %][%= a.highdst %][% } %]];
  [% } %]
  [% } %]
  }
[% } %]
[% for (a in MetaAssociation.allInstances()) { %]
  [% var src : MetaClass = MetaClasses.select(n | n.properties.includes(a)).first(); %]
  Edge [%= a %] ([%= src %]Type.[%= a %]_outgoing, [%= a.target %]Type.[%= a %]_incoming) {}
[% } %]
}
```

Listing C.3: `CompileMM.egl`: code generation of a *metaDepth* metamodel.

```
// generated

Model ElevatorDSML {
  Node FloorType {
    nr : int;
    next_outgoing : FloorType[0..1];
    next_incoming : FloorType[0..1];
    requests_incoming : ButtonType[0..*];
    currentfloor_incoming : ElevatorType[0..*];
  }
  abstract Node ButtonType {
    pressed : boolean;
    requests_outgoing : FloorType[1..1];
  }
  Node ElevatorButtonType : ButtonType {
    elevator_button_incoming : ElevatorType[1..1];
  }
  Node ElevatorType {
    doors_open : boolean;
    going_up : boolean;
    currentfloor_outgoing : FloorType[1..1];
    elevator_button_outgoing : ElevatorButtonType[0..*];
  }
  abstract Node FloorButtonType : ButtonType {
  }
  Node UpButtonType : FloorButtonType {
  }
  Node DownButtonType : FloorButtonType {
  }
  Edge next (FloorType.next_outgoing, FloorType.next_incoming) {}
  Edge elevator_button (ElevatorType.elevator_button_outgoing, ElevatorButtonType.elevator_button_incoming) {}
  Edge requests (ButtonType.requests_outgoing, FloorType.requests_incoming) {}
  Edge currentfloor (ElevatorType.currentfloor_outgoing, FloorType.currentfloor_incoming) {}
}
```

Listing C.4: `ElevatorDSML.mdepth`: the generated *metaDepth* metamodel.

```
load "ElevatorDSML"
load "ElevatorEvo"
ElevatorDSML ThreeFloors imports ElevatorEvo {
  ElevatorType e { doors_open = false; going_up = false; }
  FloorType f1 { nr = 1; }
  FloorType f2 { nr = 2; }
  FloorType f3 { nr = 3; }
  ElevatorButtonType b1 { pressed = false; }
  ElevatorButtonType b2 { pressed = false; }
  ElevatorButtonType b3 { pressed = true; }
  UpButtonType u1 { pressed = false; }
  UpButtonType u2 { pressed = false; }
  DownButtonType d2 { pressed = true; }
  DownButtonType d3 { pressed = false; }
  next n1 (f1, f2) {}
  next n2 (f2, f3) {}
  currentfloor c (e, f1) {}
  elevator_button eb1 (e, b1) {}
  elevator_button eb2 (e, b2) {}
  elevator_button eb3 (e, b3) {}
  requests r1 (b1, f1) {}
  requests r2 (b2, f2) {}
  requests r3 (b3, f3) {}
  requests r4 (u1, f1) {}
  requests r5 (u2, f2) {}
  requests r6 (d2, f2) {}
  requests r7 (d3, f3) {}
}
```

Listing C.5: `ThreeFloors.mdepth`: an instance of the ElevatorDSML metamodel.

```
load "MetaModels"
Model DeltaModels imports MetaModels {
  Node DeltaModel { operations : DeltaOperation[*] {ordered}; }
  Node DeltaOperation {}
  Node GeneraliseProperty : DeltaOperation {
       prop : String; lowsrc : int; highsrc : int; lowdst : int; highdst : int;
       pre1 : $self.highsrc > 0 and self.highdst > 0$
       pre2 : $self.lowsrc <= prop.lowsrc and self.lowdst <= prop.lowdst
               and self.highsrc >= prop.highsrc and self.highdst >= prop.highdst$ }
  Node AddClass : DeltaOperation { name : String; superclass : String[0..1]; abstract_ : boolean = false;
       pre1 : $not model.exists(c | c.toString() == name)$ }
  Node ExtractAbstractSuperClass : DeltaOperation { name : String; subclasses : String[*]; }
  Node MakeClassConcrete : DeltaOperation { class : String; }
  Node PullPropertyToAbstractClass : DeltaOperation { prop : String; }
  Node EliminateClass : DeltaOperation { class : String; }
  Node EliminateProperty : DeltaOperation { prop : String; }
  Node RenameClass : DeltaOperation { class : String; name : String; }
  Node RenameProperty : DeltaOperation { prop : String; name : String; }
  Node AddObligatoryProperty : DeltaOperation {
       name : String; owner : String; lowsrc : int; highsrc : int; type : String[0..1];
       target : String[0..1]; lowdst : int[0..1]; highdst : int[0..1]; default : String[0..1]; }
  Node RestrictProperty : DeltaOperation { prop : String; lowsrc : int; highsrc : int; lowdst : int; highdst : int; }
  // ...
}
```

Listing C.6: `DeltaModels.mdepth` (partial): the metamodel for modelling metamodel differences.

```
load "DeltaModels"
load "ElevatorMM"
DeltaModels ElevatorEvo imports ElevatorMM {
  DeltaModel ElevatorEvo { operations = [deltar, delta1, delta2, delta3, delta4, delta5, delta6, delta7, delta8,
       delta9]; }
  RenameClass deltar { class = Button; name = "CallButton"; }
  RestrictProperty delta1 { prop = "requests"; lowsrc = 0; highsrc = 3; lowdst = 1; highdst = 1; }
  ExtractAbstractSuperClass delta2 { name = "Button"; subclasses = ["CallButton"]; }
  PullPropertyToAbstractClass delta3 { prop = "pressed"; }
  AddClass delta4 { name = "EmergencyButton"; superclass = "Button"; }
  MakeClassConcrete delta5 { class = "FloorButton"; }
  AddObligatoryProperty delta6 { name = "up_direction"; owner = "FloorButton"; lowsrc = 1; highsrc = 1; type = "
       BOOLEAN"; default = "False"; }
  EliminateClass delta7 { class = "UpButton"; }
  EliminateClass delta8 { class = "DownButton"; }
  EliminateProperty delta9 { prop = "nr"; }
}
```

Listing C.7: `ElevatorEvo.mdepth`: the delta model of the evolution of the *Elevator* metamodel.

```
import "Helpers.eol";
operation main() {
  // first, put all metaelements in a container so we can add new elements
  var m : MetaModel = MetaModel.allInstances().first();
  m.elements = MetaElement.allInstances();
  // rename container
  m.name("E" + m.toString());
  // apply delta operations
  ("Applying the delta operations to create the new version of the metamodel " + m).println();
  for (delta in DeltaModel.allInstances().first().operations) {
    (delta.toString() + " " + delta.type().toString()).println();
    delta.apply(m);
  }
}
operation MakeClassConcrete apply(m : MetaModel) { self.class.getClass().is_abstract := false; }
operation AddClass apply(m : MetaModel) {
  var c : MetaClass = new MetaClass;
  c.name(self.name);
  c.superclass = self.superclass.getClass();
  c.is_abstract = self.abstract_;
  m.elements.add(c); }
operation ExtractAbstractSuperClass apply(m : MetaModel) {
  var class : MetaClass = new MetaClass;
  class.name(self.name);
  class.is_abstract = true;
  for (subclass in self.subclasses) { subclass.getClass().superclass = class; }
  m.elements.add(class); }
operation EliminateClass apply(m : MetaModel) {
  var class : MetaClass = self.class.getClass();
  m.elements.remove(class);
  for (prop in class.properties.includingAll(m.elements.select(e | e.isTypeOf(MetaAssociation) and e
       .target == class))) {
    m.elements.remove(prop);
    if (prop.getOwner().isDefined()) { prop.getOwner().properties.remove(prop); }
    delete prop;
  }
  delete class;
}
operation RenameClass apply(m : MetaModel) { self.class.getClass().name(self.name); }
operation RestrictProperty apply(m : MetaModel) {
  var prop : MetaProperty = self.prop.getProperty();
  prop.lowsrc = self.lowsrc;
  prop.highsrc = self.highsrc;
  prop.lowdst = self.lowdst;
  prop.highdst = self.highdst;
}
// ...
```

Listing C.8: `Apply.eol` (partial): the transformation in EOL that applies a delta model to a metamodel.

```
operation String getClass() { // get class by name
  return MetaClass.allInstances().select(c | c.toString() == self).first(); }
operation String getProperty() { // get property by name
  return MetaProperty.allInstances().select(p | p.toString() == self).first(); }
operation MetaProperty getOwner() : MetaClass { // get owner of property
  return MetaClass.allInstances().select(c | c.properties.includes(self)).first(); }
operation MetaClass getProperties() : Bag(MetaProperty) { // return all (inherited) properties
  var classes : Sequence(MetaClass)=self.asSet().closure(c|c.superclass).includingAll(self.asSet());
  return classes.collect(c | c.properties).includingAll(MetaProperty.allInstances()
                .select(p | p.isTypeOf(MetaAssociation) and classes.includes(p.target))).flatten();}
operation MetaClass allInstances_() : Sequence(Node) { // return all instances of a given metaclass
  var classes : Sequence(String); classes.add(self);
  classes.addAll(classes.closure(c | MetaClass.allInstances()
                                              .select(s | s.superclass.asString() = c.asString())));
  return Node.allInstances().select(n | classes.collect(c | c.asString()+"Type")
                                               .includes(n.type().toString()));
}
// return all instances of a given metaproperty Bug alert! flatten() doesn't work!
//   because association end is a not-eol type [x, y, z] it cannot be flattened
//   hence an extra test (to filter these out) n.get(fieldname).type().toString() == self.asString()
//   consequently n-to-n association instances cannot be found,
//   in that case, try self.getAllInstances() instead but this method is not always available (?!)
operation MetaProperty allInstances_() : Sequence(Any) {
  var fieldname : String = self.asString();
  if (self.target == self.getOwner()) {
    fieldname = self.asString()+self.asString()+"_incoming";
  }
  return Node.allInstances().select(n | n.hasField(fieldname)
                                     and n.get(fieldname).type().toString() == self.asString())
                            .collect(n | n.get(fieldname)).flatten();
}
// remove link (needs removing of references and therefore not just "delete" should be used)
operation removeLink(link) {
  var typename : String = link.type().asString();
  // remove source references
  var src : Any = link.get(typename+"_incoming");
  if (src.hasField(typename+"_outgoing")) src.fields.remove(src.getField(typename+"_outgoing"));
  if (src.hasField(typename+"_incoming")) src.fields.remove(src.getField(typename+"_incoming"));
  // remove target references
  var tgt : Any = link.get(typename+"_outgoing");
  if (tgt.hasField(typename+"_incoming")) tgt.fields.remove(tgt.getField(typename+"_incoming"));
  if (tgt.hasField(typename+"_outgoing")) tgt.fields.remove(tgt.getField(typename+"_outgoing"));
  // remove link
  delete link;
}
// remove all instances of a given type
operation MetaElement removeInstances() {
  if (self.isTypeOf(MetaAssociation)) {
    // remove all links and their references
    for (link in self.allInstances_()) { removeLink(link); }
  } else if (self.isTypeOf(MetaClass)) {
    // remove all node links
    for (prop in self.getProperties().select(p | p.isTypeOf(MetaAssociation))) {
      // remove links from/to an instance of the given type
      for (link in prop.allInstances_().select(l |
             l.get(prop.asString()+"_outgoing").type().toString() == self.asString()+"Type"
             or l.get(prop.asString()+"_incoming").type().toString() == self.asString()+"Type")) {
        removeLink(link);
      }
    }
    // remove all instances
    for (node in self.allInstances_()) { delete node; }
  }
}
```

Listing C.9: Helpers.eol (not shown in Figure C.1): helper operations in EOL.

```
// generated
Model EElevatorDSML {
  Node FloorType {
    next_outgoing : FloorType[0..1];
    next_incoming : FloorType[0..1];
    requests_incoming : CallButtonType[0..3];
    currentfloor_incoming : ElevatorType[0..*];
  }
  Node ElevatorType {
    doors_open : boolean;
    going_up : boolean;
    currentfloor_outgoing : FloorType[1..1];
    elevator_button_outgoing : ElevatorButtonType[0..*];
  }
  abstract Node ButtonType {
    pressed : boolean;
  }
  abstract Node CallButtonType : ButtonType {
    requests_outgoing : FloorType[1..1];
  }
  Node ElevatorButtonType : CallButtonType {
    elevator_button_incoming : ElevatorType[1..1];
  }
  Node FloorButtonType : CallButtonType {
    up_direction : boolean;
  }
  Node EmergencyButtonType : ButtonType {
  }
  Edge next (FloorType.next_outgoing, FloorType.next_incoming) {}
  Edge elevator_button (ElevatorType.elevator_button_outgoing,
                        ElevatorButtonType.elevator_button_incoming) {}
  Edge requests (CallButtonType.requests_outgoing, FloorType.requests_incoming) {}
  Edge currentfloor (ElevatorType.currentfloor_outgoing, FloorType.currentfloor_incoming) {}
}
```

Listing C.10: `EElevatorDSML.mdepth`: the evolved DSML, result of applying ElevatorEvo to ElevatorMM, followed by compiling the metamodel by Listing C.3.

```
// Execute in a DeltaModels context, with imported MetaModels
import "Helpers.eol";
operation main() {
  // first, put all metaelements in a container so we can add new elements
  var m : MetaModel = MetaModel.allInstances().first();
  m.elements = MetaElement.allInstances();
  // rename container
  m.name("IM" + m.toString());
  ("Merging the delta operations to create the intermediate metamodel " + m).println();
  // apply rename operations
  for (delta in DeltaModel.allInstances().first().operations
                          .select(o | o.isTypeOf(RenameClass) or o.isTypeOf(RenameProperty))) {
    (delta.toString() + " " + delta.type().toString()).println();
    delta.apply(m);
  }
  // apply delta operations
  for (delta in DeltaModel.allInstances().first().operations
                      .select(o | not o.isTypeOf(RenameClass) and not o.isTypeOf(RenameProperty))) {
    (delta.toString() + " " + delta.type().toString()).println();
    delta.merge(m);
  }
}
operation RenameClass apply(m : MetaModel) { self.class.getClass().name(self.name); }
operation RenameProperty apply(m : MetaModel) { self.prop.getProperty().name(self.name); }
operation MakeClassConcrete merge(m : MetaModel) { self.class.getClass().is_abstract := false; }
operation AddClass merge(m : MetaModel) {
  var class : MetaClass = new MetaClass;
  class.name(self.name);
  class.superclass = self.superclass.getClass();
  class.is_abstract = self.abstract_;
  m.elements.add(class); }
operation ExtractAbstractSuperClass merge(m : MetaModel) {
  var class : MetaClass = new MetaClass;
  class.name(self.name);
  class.is_abstract = true;
  for (s in self.subclasses) { s.getClass().superclass = class; }
  m.elements.add(class); }
operation PullPropertyToAbstractClass merge(m : MetaModel) {
  var prop : MetaProperty = self.prop.getProperty();
  var owner : MetaClass = prop.getOwner();
  owner.properties.remove(prop);
  owner.superclass.properties = new Bag(MetaProperty)
                              .includingAll(owner.superclass.properties).including(prop); }
operation EliminateClass merge(m : MetaModel) {}
operation EliminateProperty merge(m : MetaModel) { self.prop.getProperty().lowsrc = 0; }
operation RestrictProperty merge(m : MetaModel) {}
operation AddObligatoryProperty merge(m : MetaModel) {
  var prop : MetaAttribute = new MetaAttribute;
  prop.name(self.name);
  self.owner.getClass().properties = new Bag(MetaProperty)
              .includingAll(self.owner.getClass().properties).including(prop);
  prop.lowsrc = 0;
  prop.highsrc = self.highsrc;
  prop.type = self.type;
  m.elements.add(prop);
}
// ...
```

Listing C.11: `Intermediate.eol` (partial): the transformation that creates the intermediate metamodel from a metamodel and a delta model.

```
// generated
Model IElevatorDSML {
  Node FloorType {
    nr : int[0..1];
    next_outgoing : FloorType[0..1];
    next_incoming : FloorType[0..1];
    requests_incoming : CallButtonType[0..*];
    currentfloor_incoming : ElevatorType[0..*];
  }
  Node ElevatorType {
    doors_open : boolean;
    going_up : boolean;
    currentfloor_outgoing : FloorType[1..1];
    elevator_button_outgoing : ElevatorButtonType[0..*];
  }
  abstract Node ButtonType {
    pressed : boolean;
  }
  abstract Node CallButtonType : ButtonType {
    requests_outgoing : FloorType[1..1];
  }
  Node ElevatorButtonType : CallButtonType {
    elevator_button_incoming : ElevatorType[1..1];
  }
  Node FloorButtonType : CallButtonType {
    up_direction : boolean[0..1];
  }
  Node UpButtonType : FloorButtonType {}
  Node DownButtonType : FloorButtonType {}
  Node EmergencyButtonType : ButtonType {}
  Edge next (FloorType.next_outgoing, FloorType.next_incoming) {}
  Edge elevator_button (ElevatorType.elevator_button_outgoing,
                        ElevatorButtonType.elevator_button_incoming) {}
  Edge requests (CallButtonType.requests_outgoing, FloorType.requests_incoming) {}
  Edge currentfloor (ElevatorType.currentfloor_outgoing, FloorType.currentfloor_incoming) {}
}
```

Listing C.12: `IElevatorDSML.mdepth`: the intermediate *Elevator* metamodel.

```
Merging the delta operations to create the intermediate metamodel IElevatorDSML
delta1 RestrictProperty
delta2 ExtractAbstractSuperClass
delta3 PullPropertyToAbstractClass
delta4 AddClass
delta5 MakeClassConcrete
delta6 AddObligatoryProperty
delta7 EliminateClass
delta8 EliminateClass
delta9 EliminateProperty
```

Listing C.13: Output trace of the Intermediate transformation.

```
// generated by generated Python script
load "IElevatorDSML"
load "ElevatorEvo"
IElevatorDSML ThreeFloorsI imports ElevatorEvo {
  ElevatorType e { doors_open = false; going_up = false; }
  FloorType f1 { nr = 1; }
  FloorType f2 { nr = 2; }
  FloorType f3 { nr = 3; }
  ElevatorButtonType b1 { pressed = false; }
  ElevatorButtonType b2 { pressed = false; }
  ElevatorButtonType b3 { pressed = true; }
  UpButtonType u1 { pressed = false; }
  UpButtonType u2 { pressed = false; }
  DownButtonType d2 { pressed = true; }
  DownButtonType d3 { pressed = false; }
  next n1 (f1, f2) {}
  next n2 (f2, f3) {}
  currentfloor c (e, f1) {}
  elevator_button eb1 (e, b1) {}
  elevator_button eb2 (e, b2) {}
  elevator_button eb3 (e, b3) {}
  requests r1 (b1, f1) {}
  requests r2 (b2, f2) {}
  requests r3 (b3, f3) {}
  requests r4 (u1, f1) {}
  requests r5 (u2, f2) {}
  requests r6 (d2, f2) {}
  requests r7 (d3, f3) {}
}
```

Listing C.14: `ThreeFloorsI.mdepth`: the converted instance, that has been subjected to μ_r and is conform to the intermediate metamodel.

```
import "Helpers.eol";
operation main() {
  ("Migrating model").println();
  Migrate(Operations());
}
operation Operations() : Sequence(Any) { return DeltaModel.allInstances().first().operations; }
// for subscribing custom migration operations
operation Sequence(Any) Subscribe(op : String, i : Integer) {
  var fullOps : Sequence = self.select(o | not o.isTypeOf(RenameClass)
                                       and not o.isTypeOf(RenameProperty));
  var offset : Integer = self.size() - fullOps.size();
  return self.select(op | self.indexOf(op) < offset + i - 1)
             .including(op).includingAll(self.select(op | self.indexOf(op) >= offset + i - 1));
}
// do the migration
operation Migrate(operations : Sequence(Any)) {
  for (delta in operations) {
    (delta + " " + delta.type()).println();
    delta.migrate();
  }
}
operation EliminateClass migrate() {
  for (instance in self.class.getClass().allInstances_()) {
    ("removing " + instance).println();
    // go over all edges
    for (field in instance.fields.select(f | not f.isDataType()
                                         and not Node.allInstances().includes(f.get()))) {
      ("removing " + field.get()).println();
      // remove field references at target
      var assoc : MetaAssociation = field.toString().getProperty(); // find all metaproperties
      for (targetinstance in assoc.target.allInstances_()) {
        var assocvalue : Any = targetinstance.get(assoc.toString()+"_incoming");
        if (assoc.highdst <> 1) {
          assocvalue.remove(instance);
        } else {
          assocvalue.setSet(false);
        }
      }
      // remove field
      delete field.get();
    }
    delete instance;
  }
}
// ...
operation String migrate() {
  custom(self);
}
```

Listing C.15: `Migrate.eol` (partial): the transformation that implements the default migration operations.

```
import "Migrate.eol";
operation main() { ElevatorEvoM(); }
operation ElevatorEvoM() {
  ("Custom migration model").println();
  var operations : Sequence(Any) = Operations();
  operations = operations.Subscribe("migrate5_8", 5);
  operations = operations.Subscribe("migrate1", 1);
  operations.println();
  Migrate(operations);
}
operation custom(op : String) {
  if (op == "migrate1") { migrate1(); }
  else if (op == "migrate5_8") { migrate5_8(); }
}
operation migrate1() { /* do nothing */ }
operation migrate5_8() {
  for (instance in UpButton.allInstances_().includingAll(DownButton.allInstances_())) {
    var name : String = instance.toString();
    var floorbutton : FloorButtonType = new FloorButtonType;
    if (UpButton.allInstances_().includes(instance)) {
      floorbutton.up_direction = true;
    } else {
      floorbutton.up_direction = false;
    }
    floorbutton.pressed = instance.pressed;
    floorbutton.requests = instance.requests;
    delete instance;
    floorbutton.name(name);
  }
}
```

Listing C.16: `ElevatorEvoM.eol`: The manually adapted migration operations.

```
Custom migration model
Sequence {deltar, migrate1, delta1, delta2, delta3, delta4, migrate5_8, delta5, delta6, delta7, delta8, delta9}
deltar RenameClass
deltar ignored
migrate1 org.eclipse.epsilon.eol.types.EolPrimitiveType@1ed828d
delta1 RestrictProperty
delta2 ExtractAbstractSuperClass
delta2 ignored
delta3 PullPropertyToAbstractClass
delta3 ignored
delta4 AddClass
delta4 ignored
migrate5_8 org.eclipse.epsilon.eol.types.EolPrimitiveType@1ed828d
delta5 MakeClassConcrete
delta5 ignored
delta6 AddObligatoryProperty
delta7 EliminateClass
delta8 EliminateClass
delta9 EliminateProperty
removing nr in f1
removing nr in f2
removing nr in f3
```

Listing C.17: Output trace of the Migration transformation.

```
// generated by generated Python script
load "EElevatorDSML"
EElevatorDSML MThreeFloors {
  ElevatorType e { doors_open=false; going_up=false; }
  FloorType f1 {}
  FloorType f2 {}
  FloorType f3 {}
  ElevatorButtonType b1 { pressed=false; }
  ElevatorButtonType b2 { pressed=false; }
  ElevatorButtonType b3 { pressed=true; }
  FloorButtonType u1 { pressed=false; up_direction=true; }
  FloorButtonType u2 { pressed=false; up_direction=true; }
  FloorButtonType d2 { pressed=true; up_direction=false; }
  FloorButtonType d3 { pressed=false; up_direction=false; }
  next n1(f1,f2) {}
  next n2(f2,f3) {}
  currentfloor c(e,f1) {}
  elevator_button eb1(e,b1) {}
  elevator_button eb2(e,b2) {}
  elevator_button eb3(e,b3) {}
  requests r1(b1,f1) {}
  requests r2(b2,f2) {}
  requests r3(b3,f3) {}
  requests r4(u1,f1) {}
  requests r5(u2,f2) {}
  requests r6(d2,f2) {}
  requests r7(d3,f3) {}
}
```

Listing C.18: `MThreeFloors.mdepth`: the final migrated instance after the migration and conversion step, conform to the EElevatorDSML metamodel.

```
// ...
operation migrate9inv() {
  var current : FloorType = Floor.allInstances_().select(f | not f.next_incoming.isDefined()).first
      ();
  var n : Integer = 1;
  while (current.isDefined()) {
    current.nr = n;
    n = n+1;
    current = current.next_outgoing;
  }
}
operation migrate5_8inv() {
  for (instance in FloorButton.allInstances_()) {
    var name : String = instance.toString();
    var floorbutton : FloorButtonType;
    if (instance.up_direction) {
      floorbutton = new UpButtonType;
    } else {
      floorbutton = new DownButtonType;
    }
    floorbutton.pressed = instance.pressed;
    floorbutton.requests = instance.requests;
    delete instance;
    floorbutton.name(name);
  }
}
```

Listing C.19: `ElevatorEvoInv.eol`: the manually created inverse migration operations.

```
import "Helpers.eol";
@model(potency=0)
operation main() { ElevatorSim(); }
operation ElevatorSim() {
  var e : ElevatorType = Elevator.allInstances_().first(); // only one elevator
  while (pressedButtons().size() > 0) {
    var currentFloor : FloorType = e.currentfloor.currentfloor_outgoing;
    printState();
    // open door
    if (not e.doors_open and pressedButtons().select(b |
             b.requests.requests_outgoing = currentFloor and
             (b.type().asString() = "ElevatorButtonType"
              or (b.type().asString() = "UpButtonType" and e.going_up)
              or (b.type().asString() = "DownButtonType" and not e.going_up))).size() > 0) {
      "open door".println();
      for (button in pressedButtons().select(b | b.requests.requests_outgoing = currentFloor)) {
        if (not e.doors_open) e.doors_open := true;
        button.pressed := false;
      }
    // close door
    } else if (e.doors_open and pressedButtons().select(b |
                       b.requests.requests_outgoing <> currentFloor).size() > 0) {
      "close door".println();
      e.doors_open := false;
    // move
    } else if (not e.doors_open and e.going_up and pressedButtons().select(b |
                           b.requests.requests_outgoing.nr > currentFloor.nr).size() > 0) {
      "move up".println();
      e.currentfloor.currentfloor_outgoing := currentFloor.next_outgoing;
    } else if (not e.doors_open and not e.going_up and pressedButtons().select(b |
                           b.requests.requests_outgoing.nr < currentFloor.nr).size() > 0) {
      "move down".println();
      e.currentfloor.currentfloor_outgoing := currentFloor.next_incoming;
    // change dir
    } else if ((e.going_up
              and not (pressedButtons().select(b |
                    b.requests.requests_outgoing.nr > currentFloor.nr).size() > 0)
              and pressedButtons().select(b |
                    b.requests.requests_outgoing.nr <= currentFloor.nr).size() > 0)
             or (not e.going_up
              and not (pressedButtons().select(b |
                    b.requests.requests_outgoing.nr < currentFloor.nr).size() > 0)
              and pressedButtons().select(b |
                    b.requests.requests_outgoing.nr >= currentFloor.nr).size() > 0)) {
      "change dir".println();
      e.going_up := not e.going_up;
    } else {
      break;
    }
  }
  "DONE".println();
}
operation pressedButtons() : Sequence(ButtonType) {
  return Button.allInstances_().select(b | b.pressed);
}
```

Listing C.20: `ElevatorSim.eol`: the operational semantics transformation.

```
close door
change dir
move up
move up
open door
close door
change dir
move down
open door
DONE
```

Listing C.21: The execution trace of transforming ThreeFloorsI with ElevatorSim.

```
// generated by generated Python script
load "EElevatorDSML"
EElevatorDSML MThreeFloors {
  ElevatorType e { doors_open=true; going_up=false; }
  FloorType f1 {}
  FloorType f2 {}
  FloorType f3 {}
  ElevatorButtonType b1 { pressed=false; }
  ElevatorButtonType b2 { pressed=false; }
  ElevatorButtonType b3 { pressed=false; }
  FloorButtonType u1 { pressed=false; up_direction=true; }
  FloorButtonType u2 { pressed=false; up_direction=true; }
  FloorButtonType d2 { pressed=false; up_direction=false; }
  FloorButtonType d3 { pressed=false; up_direction=false; }
  next n1(f1,f2) {}
  next n2(f2,f3) {}
  currentfloor c(e,f2) {}
  elevator_button eb1(e,b1) {}
  elevator_button eb2(e,b2) {}
  elevator_button eb3(e,b3) {}
  requests r1(b1,f1) {}
  requests r2(b2,f2) {}
  requests r3(b3,f3) {}
  requests r4(u1,f1) {}
  requests r5(u2,f2) {}
  requests r6(d2,f2) {}
  requests r7(d3,f3) {}
}
```

Listing C.22: `MTThreeFloors.mdepth`: the transformed and migrated instance model.

Bibliography

[1] Marcus Alanen and Ivan Porres. Difference and union of models. *UML '03: The Unified Modeling Language*, pages 2–17, 2003. 3.2.1, 3.5.1

[2] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *IJWIS*, 5(3):271–304, 2009. 3.5.1

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994. 4.2.2

[4] Charles André, Frédéric Mallet, and Robert De Simone. Modeling Time(s). In *MoDELS'07*, volume LNCS 4735, pages pp. 559–573. Springer, 2007. 4.3.1

[5] Charles André, Frédéric Mallet, and Marie-Agnès Peraldi-Frati. A multiform time approach to real-time system modeling: Application to an automotive system. In *SIES'07*, pages 234–241, Lisbon, Portugal, 2007. 3

[6] Charles Ashbacher. "The Unified Modeling Language Reference Manual, Second Edition", by James Rumbaugh. *Journal of Object Technology*, 3(10):193–195, 2004. 1.2.1

[7] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. 2.2.2, 2.3.5

[8] Mira Balaban and Martin Gogolla, editors. *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015, Ottawa, Canada, September 29, 2015*, volume 1503 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015. C

[9] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *ACM Special Interest Group on Management Of Data*, 16(3):311–322, 1987. 3.5.2

[10] Mikaël Barbero, Frédéric Jouault, Jeff Gray, and Jean Bézivin. A practical approach to model extension. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *Model Driven Architecture- Foundations and Applications, Third European Conference, ECMDA-FA 2007, Haifa, Israel, June 11-15, 2007, Proccedings*, volume 4530 of *Lecture Notes in Computer Science*, pages 32–42. Springer, 2007. 4.4.1

[11] Jiri Barnat. Quo vadis explicit-state model checking. In Giuseppe F. Italiano, Tiziana Margaria-Steffen, Jaroslav Pokorný, Jean-Jacques Quisquater, and Roger Wattenhofer, editors, *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings*, volume 8939 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2015. 2.6

[12] Bruno Barroca, Thomas Kühne, and Hans Vangheluwe. Integrating language and ontology engineering. In *Proceedings of the 8th International Workshop on Multi-Paradigm Modeling (MPM'14)*, volume 1237 of *CEUR Workshop Proceedings*, pages 77–86, 2014. 3.1, 5, 3.6, 5.2.1

[13] Francesco Basciani, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Automated chaining of model transformations with incompatible metamodels. In Jürgen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfrán, editors, *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, volume 8767 of *Lecture Notes in Computer Science*, pages 602–618. Springer, 2014. 3.4.2

[14] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995. 2.8

[15] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004. 4.2.2

[16] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center. 2.7.2

[17] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model transformations? transformation models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006. 1.2.2

[18] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. OCL meets CTL: towards ctl-extended OCL model checking. In Jordi Cabot, Martin Gogolla, István Ráth, and Edward D. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering*

Languages and Systems (MODELS 2013), Miami, USA, September 30, 2013., volume 1092 of *CEUR Workshop Proceedings*, pages 13–22. CEUR-WS.org, 2013. 2.7.2

[19] Xavier Blanc, Alix Mougenot, Isabelle Mounier, and Tom Mens. Incremental detection of model inconsistencies based on model operations. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings*, volume 5565 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2009. 3.2.1

[20] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 511–520. ACM, 2008. 3.2.1

[21] Pere Bonet, Catalina Lladó, Ramon Puigjaner, and William J. Knottenbelt. PIPE v2.5: A Petri Net Tool for Performance Modelling. In *23rd Latin American Conference on Informatics (CLEI 2007)*, September 2007. 1.3

[22] F. Boulanger, C. Hardebolle, C. Jacquet, and I. Prodan. Modeling time for the execution of heterogeneous models. Technical report 2013-09-03-DI-FBO, Supélec E3S, 2012. 4.3.1, 4.5

[23] Frédéric Boulanger and Cécile Hardebolle. Simulation of multi-formalism models with modhel'x. In *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008*, pages 318–327. IEEE Computer Society, 2008. (document), 4.1, 4.3.1, 4.3.1, 4.4, 4.3.3, 4.4.2, 5.1.3

[24] Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Dominique Marcadet. Semantic adaptation for models of computation. In Benoît Caillaud, Josep Carmona, and Kunihiko Hiraishi, editors, *11th International Conference on Application of Concurrency to System Design, ACSD 2011, Newcastle Upon Tyne, UK, 20-24 June, 2011*, pages 153–162. IEEE Computer Society, 2011. 2.3.4, 4.3.1

[25] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987. 1.1, 5.1

[26] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards scenario-based testing of UML diagrams. In Achim D. Brucker and Jacques Julliand, editors, *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, volume 7305 of *Lecture Notes in Computer Science*, pages 149–155. Springer, 2012. 2.7.1

[27] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, volume 7320 of *Lecture Notes in Computer Science*, pages 336–398. Springer, 2012. 3.5.1

[28] P. Caspi, A. Benveniste, R. Lublinerman, and S. Tripakis. Actors without directors: A kahnian view of heterogeneous systems. In R. Majumdar and P. Tabuada, editors, *Hybrid Systems: Computation and Control*, volume 5469 of *LNCS*, pages 46–60. Springer Berlin Heidelberg, 2009. 4.3.1

[29] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. A solution for concurrent versioning of metamodels and models. *Journal of Object Technology*, 11(3):1: 1–32, 2012. 3.5.1

[30] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007. 3.1, 3.2.1, 3.5.1, 5.1.2

[31] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pages 222–231. IEEE Computer Society, 2008. (document), 3.2.1, 3.2.1, 3, 1, 3.1, 2, 3.5.2

[32] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. Proving and explaining the unfeasibility of message sequence charts for hybrid systems. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 54–62. FMCAD Inc., 2011. 2.7.1

[33] Tony Clark, Andy Evans, and Stuart Kent. Aspect-oriented metamodelling. *The Computer Journal*, 46:566–577, 2003. 4.4.1

[34] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. 2.1, 5.1.1

[35] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. 2.7.2

[36] Benoît Combemale, Xavier Crégut, and Marc Pantel. A design pattern to build executable dsmls and associated v&v tools. In Karl R. P. H. Leung and Pornsiri Muenchaisri, editors, *19th Asia-Pacific Software Engineering Conference, APSEC 2012, Hong Kong, China, December 4-7, 2012*, pages 282–287. IEEE, 2012. 2.7.2

[37] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *J. Vis. Lang. Comput.*, 13(6):573–600, 2002. 1.2.1, 1.2.5, 4.3.3

[38] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola WEAVR: aspect and model-driven engineering. *Journal of Object Technology*, 6(7):51–88, 2007. 4.1, 4.4.1

[39] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA - visual automated transformations for formal verification and validation of UML models. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 267–270. IEEE Computer Society, 2002. 5.2.3

[40] Jesús Sánchez Cuadrado, Juan de Lara, and Esther Guerra. Bottom-up meta-modelling: An interactive approach. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2012. 3.3, 3.4.2

[41] Simone André da Costa Cavalheiro, Luciana Foss, and Leila Ribeiro. Specification patterns for properties over reachable states of graph grammars. In Rohit Gheyi and David A. Naumann, editors, *Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings*, volume 7498 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012. 2.7.2

[42] Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2010. 1.2.5, 4.1, 4.2.1, 4.3.4

[43] Juan de Lara and Esther Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In *ECMFA'12*, volume 7349 of *LNCS*, pages 259–274. Springer, 2012. 1.2.5, 4.2.4, 4.2.4, 4.3.4

[44] Juan de Lara and Esther Guerra. From types to type requirements: Genericity for model-driven engineering. *Softw. Syst. Model.*, 12(3):453–474, July 2013. (document), 4.1, 4.2, 4.2.1, 4.1, 4.2.1

[45] Juan de Lara, Tihamer Levendovszky, and Pieter J. Mosterman. Guest editorial: Special issue on multi-paradigm modeling. *Simulation*, 85(11-12):685–687, 2009. (document), 1.1, 5.1

[46] Juan De Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism modelling and meta-modelling. *Lecture Notes in Computer Science*, 2306:174–188, 2002. 1.1, 1.2.5, 3.3

[47] Juan de Lara and Hans Vangheluwe. Using AToM^3 as a meta-case tool. In *ICEIS*, pages 642–649, 2002. 1.2.5

[48] Joachim Denil, Bart Meyers, Bart Pussig, Paul De Meulenaere, and Hans Vangheluwe. Explicit semantic adaptation of hybrid formalisms for fmi co-simulation. In *Proceedings of the 2015 Symposium on Theory of Modeling and Simulation - DEVS, part of the Spring Simulation Multi-Conference, TMS/DEVS '15*, pages 852–859, 2015. 4.3.3

[49] Romuald Deshayes, Bart Meyers, Tom Mens, and Hans Vangheluwe. Promobox in practice : A case study on the GISMO domain-specific modelling language. In Daniel Balasubramanian, Christophe Jacquet, Pieter Van Gorp, Sahar Kokaly, and Tamás Mészáros,

editors, *Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, MPM@MODELS 2014, Valencia, Spain, September 30, 2014.*, volume 1237 of *CEUR Workshop Proceedings*, pages 21–30. CEUR-WS.org, 2014. 2.3.1, 2.3.5, 2.5, 2.6.1, 2.6.3

[50] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Traceability visualization in metamodel change impact detection. In *Proceedings of the Second Workshop on Graphical Modeling Language Development*, GMLD '13, pages 51–62, New York, NY, USA, 2013. ACM. 3.5.1

[51] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Supporting users to manage breaking and unresolvable changes in coupled evolution. In *The 15th Workshop on Domain-Specific Modeling (DSM2015) at SPLASH 2015*, 2015. 3.5.2

[52] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. Developing next generation adls through mde techniques. In *ICSE'10 (1)*, pages 85–94. ACM, 2010. 4.4.1

[53] Jürgen Dingel, Zinovy Diskin, and Alanna Zito. Understanding and improving uml package merge. *Software and System Modeling*, 7(4):443–467, 2008. 4.2.1

[54] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., 1999. 4.4.1

[55] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420. ACM, 1999. (document), 2.1, 2, 2.6.3, 2.6.3, 2.6.3, 2.7.2

[56] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003. 4.3.4, 4.4.2, 5.1.3

[57] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. MIT Press Cambridge, 1990. 2.2.1

[58] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982. 1.1, 2.1, 2.2.1, 5.1.1

[59] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *OOPSLA 6th Workshop on Domain Specific Modeling*, pages 123–139, October 2006. 4.4.1

[60] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Procs. of Workshop on Language Descriptions, Tools and Applications (LDTA)*, 2012. to appear. 4.4.1

[61] Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee, and Jianwei Niu. Deconstructing the semantics of big-step modelling languages. *Requir. Eng.*, 15(2):235–265, 2010. 5.2.1

[62] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98, Paderborn, Germany, November 16-20, 1998, Selected Papers*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998. 2.7.2

[63] Franck Fleurey, Benoit Baudry, Robert B. France, and Sudipto Ghosh. A generic approach for automatic model composition. In Holger Giese, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer, 2007. 4.1, 4.4.1

[64] Robert B. France, James M. Bieman, Sai Pradeep Mandalaparty, Betty H. C. Cheng, and Adam C. Jensen. Repository for model driven development (remodd). In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1471–1472. IEEE, 2012. 5.2.3

[65] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. *CoRR*, abs/1409.6620, 2014. (document), 1.1, 2.1, 5.1.1

[66] Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. A classification of model checking-based verification approaches for software models. In *Second Workshop on Verification Of Model Transformations (VOLT)*, 2013. 2.7.1

[67] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, volume 5562 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2009. 3.5.2

[68] Kelly Garcés, Juan M. Vara, Frédéric Jouault, and Esperanza Marcos. Adapting transformations to metamodel changes via external transformation composition. *Software and System Modeling*, 13(2):789–806, 2014. 3.5.2

[69] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 115–134. ACM, 2003. 4.1

[70] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, 2007. 4.1

[71] R. Gascon, F. Mallet, and J. Deantoni. Logical time and temporal logics: comparing UML MARTE/CCSL and PSL. In *TIME'11*, pages 141–148, Lubeck, Germany, 2011. 4.3.1

[72] Leif Geiger and Albert Zündorf. Tool modeling with Fujaba. *Electronic Notes in Theoretical Computer Science*, 148(1):173–186, February 2006. 3.3

[73] Holger Giese, Tihamer Levendovszky, and Hans Vangheluwe. Summary of the workshop on multi-paradigm modeling: Concepts and tools. In Kühne [109], pages 252–262. 1.2.1

[74] Jeff Gray, Sandeep Neema, Juha-Pekka Tolvanen, Aniruddha S. Gokhale, Steven Kelly, and Jonathan Sprinkle. Domain-specific modeling. In Paul A. Fishwick, editor, *Handbook of Dynamic System Modeling*. Chapman and Hall/CRC, 2007. (document), 1.1, 2.1, 5.1

[75] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA*, pages 291–310. ACM, 2006. 4.1, 4.2.1, 5.1.3

[76] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006. 2.8

[77] Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards synchronizing models with evolving metamodels. In *International Workshop on Model-Driven Software Evolution at IEEE European Conference on Software Maintenance and Reengineering (ECSMR)*, 2007. `http://www.sciences.univ-nantes.fr/MoDSE2007/`. 3.2.1, 3, 1, 3.5.2, 7

[78] Esther Guerra and Juan de Lara. Model view management with triple graph transformation systems. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2006. 3.3

[79] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. A visual specification language for model-to-model transformations. In Christopher D. Hundhausen, Emmanuel Pietriga, Paloma Díaz, and Mary Beth Rosson, editors, *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2010, Leganés-Madrid, Spain, 21-25 September 2010, Proceedings*, pages 119–126. IEEE Computer Society, 2010. 3

[80] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46, 2013. 3

[81] Cécile Hardebolle and Frédéric Boulanger. Exploring multi-paradigm modeling techniques. *SIMULATION* , 85(11/12):688–708, 2009. 4.3.1, 4.4.2

[82] Regina Hebig and Holger Giese. On the complex nature of mde evolution and its impact on changeability. *Software and Systems Modeling*, pages 1–24, 2015. 3.5.2

[83] Regina Hebig, Holger Giese, Florian Stallmann, and Andreas Seibel. On the complex nature of MDE evolution. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 436–453. Springer, 2013. 3.5.2

[84] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT '02: First International Conference on Graph Transformation*, pages 161–176, London, UK, 2002. Springer-Verlag. 3.2.9

[85] Jan Heering and Ralf Lämmel. Coupled software transformations. In *SET 2004, First Int. Workshop on Software Evolution Transformations*, pages 31–35, 2004. 3.5.2

[86] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope - automating coupled evolution of metamodels and models. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 52–76, 2009. (document), 3.1, 3.2.1, 3, 3.1, 5, 3.3.2, 3.5.1, 3.5.2, 5.1.2, B

[87] Markus Herrmannsdoerfer and Maximilian Koegel. Towards a generic operation recorder for model evolution. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 76–81, New York, NY, USA, 2010. ACM. 3.5.1

[88] Joachim Hoessler, Joachim Soden, Michael, and Hajo Eichler. Coevolution of models, metamodels and transformations. *Models and Human Reasoning*, pages 129–154, 2005. 3.5.2

[89] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. (document), 2.1, 2.2.2, 2.3.1, 2.5

[90] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus: A tool for distributed systems specification. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, FTRTFT'96, Uppsala, Sweden, September 9-13, 1996, Proceedings*, volume 1135 of *Lecture Notes in Computer Science*, pages 467–470. Springer, 1996. 1.3

[91] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. 2.6.1

[92] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008. 3.5.2

[93] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, CMU, November 1990. 3.4

[94] Bilal Kanso and Safouan Taha. Temporal constraint support for OCL. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 83–103. Springer, 2012. 2.7.2

[95] Petra Kaufmann, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. The past, present, and future of model versioning. In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 410–443. IGI Global, 2011. 3.5.1

[96] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* John Wiley & Sons, March 2008. 1.1, 1.2.1, 3.1, 5.1.2, 5.2.4

[97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. 4.2.6

[98] Jörg Kienzle, Wisam Al Abed, Franck Fleurey, Jean-Marc Jézéquel, and Jacques Klein. Aspect-oriented design with reusable aspect models. *T. Aspect-Oriented Software Development*, 7:272–320, 2010. 4.4.1

[99] Doug Kimelman, Marsha Kimelman, David Mandelin, and Daniel M. Yellin. Bayesian approaches to matching architectural diagrams. *IEEE Trans. Software Eng.*, 36(2):248–274, 2010. 3.5.1

[100] Florian Klein and Holger Giese. Joint structural and temporal property specification using timed story scenario diagrams. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2007. 2.1, 2, 2.7.2

[101] Jacques Klein, Loïc Hélouët, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In Robert E. Filman, editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*, pages 27–38. ACM, 2006. 4.1, 4.4.1

[102] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005. 3.5.2

[103] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking - timed UML state machines and collaborations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002, Proceedings*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002. 2.7.1

[104] Alexander Knapp and Jochen Wuttke. Model checking of UML 2.0 interactions. In Kühne [109], pages 42–51. 2.7.1

[105] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006. 1.2.5

[106] Patrick Könemann. Capturing the intention of model changes. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, volume 6395 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2010. 3.5.1

[107] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 372–381. ACM, 2005. 2.8

[108] Thomas Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006. 1.1, 1.2.1

[109] Thomas Kühne, editor. *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *Lecture Notes in Computer Science*. Springer, 2007. C

[110] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In Sudipto Ghosh, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2009. 1.2.2, 2, 1.2.2, 1.3, 2, 2.3.3, 2.3.3, 3.3

[111] Dilshodbek Kuryazov, Jan Jelschen, and Andreas Winter. Describing modeling deltas by model transformation. *Softwaretechnik-Trends*, 32(4), 2012. 3.5.1

[112] Dilshodbek Kuryazov and Andreas Winter. Representing model differences by delta operations. In Georg Grossmann, Sylvain Hallé, Dimka Karastoyanova, Manfred Reichert, and Stefanie Rinderle-Ma, editors, *18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOC Workshops 2014, Ulm, Germany, September 1-2, 2014*, pages 211–220. IEEE Computer Society, 2014. 3.5.1

[113] Ralf Lämmel. Grammar Adaptation. In *Formal Methods Europe (FME) 2001*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer-Verlag, 2001. 3.5.2

[114] Ralf Lämmel and Wolfgang Lohmann. Format Evolution. In *7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155, pages 113–134. OCG, 2001. 3.5.2

[115] A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti. On metamodel composition. In *Control Applications, 2001. (CCA '01). Proceedings of the 2001 IEEE International Conference on*, pages 756–760, 2001. 4.4.1

[116] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, and Gabor Karsai. A Novel Approach to Semi-Automated Evolution of DSML Model Transformation. In *2nd International Conference on Software Language Engineering (SLE 2009)*, volume 5969 of *Lecture Notes in Computer Science*, pages 23–41, May 2010. 3.5.2

[117] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, Feng Shi, Christopher P. van Buskirk, and Gabor Karsai. A semi-formal description of migrating domain-specific models with evolving domains. *Software and System Modeling*, 13(2):807–823, 2014. 3.5.2

[118] Xuandong Li, Jun Hu, Lei Bu, Jianhua Zhao, and Guoliang Zheng. Consistency checking of concurrent models for scenario-based specifications. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *SDL 2005: Model Driven, 12th International SDL Forum, Grimstad, Norway, June 20-23, 2005, Proceedings*, volume 3530 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2005. 2.7.1

[119] Xue Li. A survey of schema evolution in object-oriented databases. In *TOOLS '99: 31st International Conference on Technology of Object-Oriented Language and Systems*, page 362. IEEE Computer Society, 1999. 3.5.2

[120] Yuehua Lin, Jeff Gray, and Frédéric Jouault. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, August 2007. 2.6.4, 3.4.4, 3, 3.5.1

[121] Ernst Lippe and Norbert van Oosterom. Operation-based merging. In Ian Thomas, editor, *SDE 5: 5th ACM SIGSOFT Symposium on Software Development Environments, Washington, DC, USA, December 9-11, 1992*, pages 78–87. ACM, 1992. 3.2.1

[122] Qichao Liu, Marjan Mernik, and Barrett R. Bryant. Mmdiff: a modeling tool for metamodel comparison. In Randy K. Smith and Susan V. Vrbsky, editors, *Proceedings of the 50th Annual Southeast Regional Conference, 2012, Tuscaloosa, AL, USA, March 29-31, 2012*, pages 118–123. ACM, 2012. 3.5.1

[123] Yan Liu, Steffen Zschaler, Benoit Baudry, Sudipto Ghosh, Davide Di Ruscio, Ethan K. Jackson, and Manuel Wimmer, editors. *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013*, volume 1115 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014. C

[124] Levi Lücio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, GehanM.K. Selim, Eugene Syriani, and Manuel Wimmer. Model transformation intents and their properties. *Software and Systems Modeling*, pages 1–38, 2014. 5

[125] Levi Lucio, Sadaf Mustafiz, Joachim Denil, Bart Meyers, and Hans Vangheluwe. The Formalism Transformation Graph as a Guide to Model Driven Engineering. Technical Report SOCS-TR2012.1, School of Computer Science, McGill University, March 2012. (document), 1.2.4

[126] Raphaël Mannadiar. *Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling*. PhD thesis, McGill University, June 2012. 1.2.5, 4.4.1

[127] The Mathworks. Model verification using simulink control design and simulink verification blocks. `http://www.mathworks.com/help/slcontrol/ug/model-verification-using-simulink-control-design-and-simulink-verification-blocks-.html`. Accessed: February 2016. 2.1

[128] The Mathworks. Simulink - simulation and model-based design. `http://www.mathworks.com/products/simulink/`. Accessed: February 2016. 1.2.1, 2.1, 2.6.2

[129] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004. 2.8

[130] Tom Mens. A state-of-the-art survey on software merging. *IEEE Trans. Software Eng.*, 28(5):449–462, 2002. 3.5.1

[131] Tom Mens and Serge Demeyer, editors. *Software Evolution*. Springer, 2008. (document), 1.1, 3.1, 5.1.2

[132] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007. 3.2.9

[133] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. In *GraMoT'05*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 125–142, March 2006. 1.2.2, 1.2.3, 3.3

[134] Stephan Merz. An introduction to model checking. In Stephan Merz and Nicolas Navet, editors, *Modeling and Verification of Real-Time Systems - Formalisms and Software Tools*, pages 81–116. ISTE Publishing, 2008. 1.3, 2.1, 2.6.1, A

[135] Bart Meyers, Joachim Denil, Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Hans Vangheluwe. A DSL for explicit semantic adaptation. In Christophe Jacquet, Daniel Balasubramanian, Edward Jones, and Tamás Mészáros, editors, *Proceedings of the 7th Workshop on Multi-Paradigm Modeling co-located with the 16th International Conference on Model Driven Engineering Languages and Systems, MPM@MoDELS 2013, Miami, Florida, September 30, 2013.*, volume 1112 of *CEUR Workshop Proceedings*, pages 47–56. CEUR-WS.org, 2013. 2.3.4, 4.4.2

[136] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A framework for generating domain-specific property languages. In *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 1–20. Springer International Publishing, 2014. 2.3.1, 2.5, 2.6.1, 1

[137] Bart Meyers and Pieter Van Gorp. Towards a hybrid transformation language: Implicit and explicit rule scheduling in story diagrams. In Uwe Assmann, Jendrik Johannes, and Albert Zündorf, editors, *Sixth International Fujaba Days*, pages 15–18, September 2008. 2

[138] Bart Meyers, Manuel Wimmer, and Hans Vangheluwe. Towards domain-specific property languages: The ProMoBox approach. In *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, pages 39–44. ACM New York, NY, USA, 2013. 2.3.1, 2.5

[139] Simon Van Mierlo. Explicitly modelling model debugging environments. In Balaban and Gogolla [8], pages 24–29. 5.2.1

[140] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963. 2.8

[141] Mark Minas. Generating meta-model-based freehand editors. *Electronic Communications of the European Association of Software Science and Technology*, 1, 2006. `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/83`. 3.3, 3.4.2

[142] Vince Molnár, András Vörös, Dániel Darvas, Tamás Bartha, and István Majzik. Component-wise incremental ltl model checking. *Formal Aspects of Computing*, pages 1–35, 2016. 2.8

[143] Daniel L. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Software Eng.*, 35(6):756–779, 2009. 5.2.2

[144] Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. A generic weaver for supporting product lines. In *Proceedings of the 13th International Workshop on Early Aspects*, EA '08, pages 11–18, New York, NY, USA, 2008. ACM. 4.4.1

[145] Pieter J. Mosterman. *Hybrid Dynamic Systems: A Hybrid Bond Graph Modeling Paradigm and its Application in Diagnosis*. PhD thesis, Vanderbilt University, 1997. 2.3.4

[146] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004. (document), 1.1, 1.2.1, 5.1

[147] Olaf Muliawan. Extending a model transformation language using higher order transformations. *Working Conference on Reverse Engineering*, pages 315–318, 2008. 2

[148] Sadaf Mustafiz, Bruno Barroca, Claudio Gomes, and Hans Vangheluwe. Towards modular language design using language fragments: The hybrid systems case study. *13TH International Conference on Information Technology, Track on Model-Driven Engineering for Cyber-Phisical Systems (ModelCyPhy)*, 2016. To be published. 4.5

[149] Sadaf Mustafiz, Joachim Denil, Levi Lucio, and Hans Vangheluwe. The FTG+PM framework for multi-paradigm modelling: an automotive case study. In Cécile Hardebolle, Eugene Syriani, Jonathan Sprinkle, and Tamás Mészáros, editors, *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM@MoDELS 2012, Innsbruck, Austria, October 1-5, 2012*, pages 13–18. ACM, 2012. 1.2.4

[150] Anantha Narayanan, Tihamer Levendovszky, Daniel Balasubramanian, and Gabor Karsai. Automatic domain model migration to manage metamodel evolution. In *Model Driven Engineering Languages and Systems*, pages 706–711. Springer, 2009. 3.5.2

[151] Object Management Group. Business Process Modeling Notation (BPMN) Version 2.0. Technical report, OMG, 2010. 3.1, 5.1.2

[152] Object Management Group. Object constraint language version 2.4. Technical report, OMG, 2014. 1.2.1, 2.2.1, 3.2.2

[153] Object Management Group. OMG Unified Modeling Language Version 2.5. Technical report, OMG, March 2015. 1.2.4, 3.1, 4.2.1, 4.4.1, 5.1.2

[154] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. *ACM Special Interest Group on Software Engineering*, 28(5):227–236, 2003. 3.5.1

[155] Luís Pedro, Vasco Amaral, and Didier Buchs. Foundations for a Domain Specific Modeling Language Prototyping Environment: A compositional approach. In *Proc. 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM)*. University of Jyväskylän, October 2008. 4.4.1

[156] Luis Pedro, Matteo Risoldi, Didier Buchs, and Vasco Amaral. Developing domain-specific modeling languages by metamodel semantic enrichment and composition: a case study. In *Procs. of the DSM:10*, pages 16:1–16:6. ACM, 2010. 4.4.1

[157] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. Composing visual syntax for domain specific languages. In Julie A. Jacko, editor, *Human-Computer Interaction. Novel Interaction Methods and Techniques, 13th International Conference, HCI International 2009, San Diego, CA, USA, July 19-24, 2009, Proceedings, Part II*, volume 5611 of *Lecture Notes in Computer Science*, pages 889–898. Springer, 2009. 4.4.1

[158] Patrizio Pelliccione, Paola Inverardi, and Henry Muccini. CHARMY: A framework for designing and verifying architectural specifications. *IEEE Trans. Software Eng.*, 35(3):325–346, 2009. 2.7.1

[159] Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. Pattern-based development of domain-specific modelling languages. In Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, editors, *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 166–175. IEEE, 2015. (document), 1.1, 4.1

[160] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981. 1.3

[161] Markus Pizka and Elmar Jurgens. Automating language evolution. In *TASE '07: First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 305–315. IEEE Computer Society, 2007. 3.5.2

[162] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. 1.1, 2.1, 2.2.1, 5.1.1

[163] Ivan Porres. Union and difference of models, 10 years later. In Liu et al. [123], pages 1–5. 3.5.1

[164] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *VLDB*, pages 826–873, 2003. 4.1, 4.4.1

[165] Erhard Rahm and Philip A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, 2006. 3.5.2

[166] Elham Ramezani, Dirk Fahland, and Wil M. P. van der Aalst. Where did I misbehave? diagnostic information in compliance checking. In Alistair P. Barros, Avigdor Gal, and Ekkart Kindler, editors, *Business Process Management - 10th International Conference, BPM 2012, Tallinn, Estonia, September 3-6, 2012. Proceedings*, volume 7481 of *Lecture Notes in Computer Science*, pages 262–278. Springer, 2012. 2.1

[167] Y. Raghu Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development I*, 3880:75–105, 2006. 4.1, 4.4.1

[168] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998. 2.1

[169] Arend Rensink. Explicit state model checking for graph grammars. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 114–132. Springer, 2008. 2.1, 2.2.2

[170] Mark Richters and Martin Gogolla. A metamodel for ocl. In *UML'99: 2nd international conference on The unified modeling language*, pages 156–171. Springer-Verlag, 1999. 3.2.1

[171] Matteo Risoldi. *A Methodology For The Development Of Complex Domain Specific Languages*. PhD thesis, University of Geneva, 2010. 1.1, 2.1, 1, 5.1.1

[172] José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal specification and analysis of domain specific models using maude. *Simulation*, 85(11-12):778–792, 2009. 2.7.2

[173] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with maude. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 2008. 2.7.2

[174] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Dealing with the coupled evolution of metamodels and model-to-text transformations. In Alfonso Pierantonio, Bernhard Schätz, and Dalila Tamzalit, editors, *Proceedings of the Workshop on Models and Evolution co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), Valencia, Spain, Sept 28, 2014.*, volume 1331 of *CEUR Workshop Proceedings*, pages 22–31. CEUR-WS.org, 2014. 3.5.2

[175] John F. Roddick. Schema evolution in database systems - an annotated bibliography. *SIGMOD Record*, 21(4):35–40, 1992. 3.5.2

[176] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with epsilon flock. In *3rd Int. Conf. on Theory and Practice of Model Transformations*, pages 184–198. Springer, 2010. 3.5.2

[177] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, volume 7562 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2012. 3.5.2

[178] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Managing the coupled evolution of metamodels and textual concrete syntax specifications. In Onur Demirörs and Oktay Türetken, editors, *39th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2013, Santander, Spain, September 4-6, 2013*, pages 114–121. IEEE, 2013. 3.5.2

[179] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. A methodological approach for the coupled evolution of metamodels and ATL transformations. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, volume 7909 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2013. 3.5.2

[180] Laurent Safa. The practice of deploying DSM Report from a Japanese appliance maker trenches. In Jeff Gray, Juha-Pekka Tolvanen, and Jonathan Sprinkle, editors, *Sixth Object-Oriented Programming, Systems, Languages and Applications Workshop on Domain-Specific Modeling*, pages 185–196. University of Jyväskylä, October 2006. (document), 1.1, 3.1, 5.1.2

[181] Jesus Sanchez Cuadrado and Jesus Garcia Molina. A model-based approach to families of embedded domain-specific languages. *IEEE Trans. Softw. Eng.*, 35(6):825–840, November 2009. 4.4.1

[182] Douglas C. Schmidt. Guest editor's introduction: Model-Driven Engineering. *IEEE Computer*, 39:25–31, 2006. (document), 1.1, 5.1

[183] Maik Schmidt and Tilman Gloetzner. Constructing difference tools for models using the SiDiff framework. In *International Conference on Software Engineering Companion '08: Companion of the 30th international conference on Software engineering*, pages 947–948. ACM, 2008. 3.5.1

[184] Andy Schürr. Specification of graph translators with triple graph grammars. In *WG '94: 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, London, UK, 1995. Springer-Verlag. 4

[185] Margaret H. Smith, Gerard J. Holzmann, and Kousha Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada*, pages 14–22. IEEE Computer Society, 2001. 2.1, 3, 2.7.1

[186] Jonathan Sprinkle, Jeffrey Gray, and Marjan Mernik. Fundamental limitations in domain-specific language evolution. Technical Report TR-090831, University of Arizona, 2009. 3.5.2

[187] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, April 2004. 1.1, 3.1, 3.5.1, 3.5.2

[188] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. 5.2.3

[189] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009. 4.1

[190] Matthew Stephan and James R. Cordy. A survey of model comparison approaches and applications. In Slimane Hammoudi, Luís Ferreira Pires, Joaquim Filipe, and Rui César das Neves, editors, *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain, 19 - 21 February, 2013*, pages 265–277. SciTePress, 2013. 3.5.1

[191] Frank Strobl and Alexander Wisspeintner. Specification of an elevator control system – an autofocus case study. Technical Report TUM-I9906, Technische Univerität München, 1999. 1.3

[192] Hong Su, Diane Kramer, Li Chen, Kajal Claypool, and Elke A. Rundensteiner. Xem: Managing the evolution of xml documents. In *RIDE ’01: 11th International Workshop on research Issues in Data Engineering*, page 103. IEEE Computer Society, 2001. 3.5.2

[193] Eugene Syriani. *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, McGill University Montreal, Canada, 2011. 2.3.3, 2.3.3

[194] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In Liu et al. [123], pages 21–25. 2.5

[195] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software and System Modeling*, 13(1):239–272, 2014. 3.5.1

[196] Yentl Van Tendeloo. Foundations of a multi-paradigm modelling tool. In Balaban and Gogolla [8], pages 52–57. 5.2.2

[197] Andrey A. Terekhov and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17:111–124, 2000. 1.1, 3.1, 5.1.2

[198] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *5th European Conf. on Model Driven Architecture - Foundations and Applications*, pages 18–33. Springer, 2009. 3.3.2

[199] Rafael Ugaz. Weaving of domain-specific modelling languages, a literature review. Technical report, University of Antwerp, 2014. 4.4.1, 4.5

[200] Rafael Ugaz. Combination of domain-specific languages. Master's thesis, University of Antwerp, 2015. 4.5

[201] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. Emf-incquery: An integrated development environment for live model queries. *Sci. Comput. Program.*, 98:80–99, 2015. 2.8

[202] Antonio Vallecillo. On the combination of domain specific modeling languages. In *ECMFA'10*, volume 6138 of *LNCS*, pages 305–320. Springer, 2010. 4.4.1

[203] Mark van den Brand, Albert Hofkamp, Tom Verhoeff, and Zvezdan Protić. Assessing the quality of model-comparison tools: A method and a benchmark data set. In *Proceedings of the 2Nd International Workshop on Model Comparison in Practice*, IWMCP '11, pages 2–11, New York, NY, USA, 2011. ACM. 3.5.1

[204] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. Fine-grained metamodel-assisted model comparison. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 11–20, New York, NY, USA, 2010. ACM. 3.5.1

[205] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. Generic tool for visualization of model differences. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 66–75, New York, NY, USA, 2010. ACM. 3.5.1

[206] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. A generic solution for syntax-driven model co-evolution. In Judith Bishop and Antonio Vallecillo, editors, *Objects, Models, Components, Patterns - 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings*, volume 6705 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2011. 3.5.2

[207] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004. 2.7.2

[208] Gergely Varró, Katalin Friedl, and Dániel Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electr. Notes Theor. Comput. Sci.*, 152:191–205, 2006. 2.6.4

[209] Sander Vermolen and Eelco Visser. Heterogeneous coupled evolution of software languages. In *MoDELS '08: 11th international conference on Model Driven Engineering Languages and Systems*, pages 630–644. Springer-Verlag, 2008. 3, 3.5.2

[210] Willem Visser, Matthew B. Dwyer, and Michael W. Whalen. The hidden models of model checking. *Software and System Modeling*, 11(4):541–555, 2012. 1.1, 2.1, 5.1.1

[211] Markus Voelter. Language and ide modularization, extension and composition with mps. In *Procs. of the GTTSE 2011 Summer School*, 2011. 4.4.1

[212] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer-Verlag, July 2007. 3.2.1, 3, 3.5.2

[213] J. White, J.H. Hill, J. Gray, S. Tambe, A.S. Gokhale, and D.C. Schmidt. Improving domain-specific language reuse with software product line techniques. *Software, IEEE*, 26(4):47 –53, 2009. 4.4.1

[214] Jon Whittle and Praveen Jayaraman. MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 16–27. Springer Berlin / Heidelberg, 2008. 4.4.1

[215] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65. ACM, 2005. 3, 3.5.1

[216] Dmitry A. Zaitsev. A small universal petri net. In Turlough Neary and Matthew Cook, editors, Proceedings *Machines, Computations and Universality 2013,* Zürich, Switzerland, 9/09/2013 - 11/09/2013, volume 128 of *Electronic Proceedings in Theoretical Computer Science*, pages 190–202. Open Publishing Association, 2013. 3.2.6

[217] Faiez Zalila, Xavier Crégut, and Marc Pantel. Leveraging formal verification tools for DSML users: A process modeling case study. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*, volume 7610 of *Lecture Notes in Computer Science*, pages 329–343. Springer, 2012. 2.7.2

[218] Vadim Zaytsev. Negotiated grammar evolution. *Journal of Object Technology*, 13(3):1: 1–22, 2014. 3.5.2

[219] Jing Zhang and Jeff Gray. A generative approach to model interpreter evolution. In *Object-Oriented Programming, Systems, Languages and Applications Workshop on Domain-Specific Modeling*, pages 121–129, 11 2004. 3.5.2

[220] Zu Zhang, Renwei Zhang, and Zheng Qin. Composite-level conflict detection in uml model versioning. *Mathematical Problems in Engineering*, 2015(650748):1–9, 2015. 3.5.1

[221] Paul Ziemann and Martin Gogolla. OCL extended with temporal logic. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers*, volume 2890 of *Lecture Notes in Computer Science*, pages 351–357. Springer, 2003. 2.7.2