UNIVERSITEIT ANTWERPEN

PHD THESIS

A Foundation for Multi-Paradigm Modelling

Een Onderbouw voor Multi-Paradigma Modelleren

Auteur:
Yentl VAN TENDELOO

Promotor:
Prof. dr. Hans VANGHELUWE

Proefschrift ingediend tot het behalen van de graad van Doctor in de Wetenschappen: Informatica

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
  - 1.1.1 Usefulness of MPM . . . 2
  - 1.1.2 Tool Support . . . 4
  - 1.1.3 Use of MPM Techniques . . . 4
  - 1.1.4 Foundation for MPM . . . 5
- 1.2 Challenges and Contributions . . . 5

2 Background **9**
- 2.1 Domain-Specific Modelling . . . 9
  - 2.1.1 Example . . . 10
  - 2.1.2 Terminology . . . 10
  - 2.1.3 Syntax . . . 11
  - 2.1.4 Semantics . . . 14
- 2.2 Process Modelling . . . 18
- 2.3 Multi-Paradigm Modelling . . . 19
- 2.4 Formalisms . . . 20
  - 2.4.1 Finite State Automata . . . 20
  - 2.4.2 Causal Block Diagrams . . . 21
  - 2.4.3 Petri Nets . . . 24
  - 2.4.4 Statecharts . . . 25
  - 2.4.5 Statecharts + Class Diagrams . . . 26
  - 2.4.6 Discrete Event System Specification . . . 27
  - 2.4.7 Formalism Transformation Graph + Process Model . . . 31

3 State of the Art **35**
- 3.1 Language Engineering . . . 35
  - 3.1.1 Instantiation and Conformance . . . 35
  - 3.1.2 Model Finding and Type Inference . . . 37
  - 3.1.3 Multiple Dimensions . . . 38
  - 3.1.4 Multi-Level Modelling . . . 39
- 3.2 Activities . . . 40
  - 3.2.1 Model Transformations . . . 40
  - 3.2.2 Procedural Code . . . 41
- 3.3 Processes . . . 41
- 3.4 Megamodelling . . . 42

3.5 Modelling as a Service . . . 43
3.6 Tool Comparison . . . 43

4 Modelverse Specification **47**
4.1 Types of Users . . . 47
4.1.1 Modeller . . . 48
4.1.2 Language Engineer . . . 48
4.1.3 Modelverse Developer . . . 49
4.2 Requirements . . . 50
4.2.1 Multi-Paradigm Modelling . . . 50
4.2.2 Kernel . . . 52
4.2.3 Repository . . . 55
4.2.4 Non-Functional Requirements . . . 56
4.3 Architecture . . . 58
4.3.1 Modelverse Interface . . . 58
4.3.2 Modelverse Kernel . . . 59
4.3.3 Modelverse State . . . 62

5 Modelverse Development using MPM **67**
5.1 Graphical User Interface (GUI) . . . 69
5.1.1 Motivation . . . 69
5.1.2 Model . . . 70
5.1.3 Evaluation . . . 72
5.1.4 Link to Requirements . . . 73
5.2 Wrapper . . . 73
5.2.1 Motivation . . . 74
5.2.2 Model . . . 74
5.2.3 Evaluation . . . 75
5.2.4 Link to Requirements . . . 76
5.3 Network Protocols . . . 76
5.3.1 Motivation . . . 77
5.3.2 Model . . . 77
5.3.3 Evaluation . . . 79
5.3.4 Link to Requirements . . . 80
5.4 Core Library . . . 80
5.4.1 Motivation . . . 81
5.4.2 Model . . . 82
5.4.3 Evaluation . . . 84
5.4.4 Link to Requirements . . . 85
5.5 Formalism Transformation Graph . . . 86
5.5.1 Motivation . . . 86
5.5.2 Model . . . 87
5.5.3 Evaluation . . . 88
5.5.4 Link to Requirements . . . 88
5.6 Conformance Algorithm . . . 89
5.6.1 Motivation . . . 89
5.6.2 Model . . . 94
5.6.3 Evaluation . . . 95

5.6.4 Related Work . . . 97
5.6.5 Link to Requirements . . . 99
5.7 Physical Type Model . . . 99
5.7.1 Motivation . . . 100
5.7.2 Model . . . 101
5.7.3 Evaluation . . . 105
5.7.4 Related Work . . . 108
5.7.5 Dynamic PTM Optimization using Activity Models . . . 109
5.7.6 Link to Requirements . . . 117
5.8 Service Orchestration . . . 118
5.8.1 Motivation . . . 119
5.8.2 Model . . . 120
5.8.3 Evaluation . . . 121
5.8.4 Related Work . . . 121
5.8.5 Link to Requirements . . . 123
5.9 FTG+PM Enactment . . . 124
5.9.1 Motivation . . . 124
5.9.2 Model . . . 125
5.9.3 Evaluation . . . 128
5.9.4 Related Work . . . 128
5.9.5 Link to Requirements . . . 130
5.10 Action Language . . . 130
5.10.1 Motivation . . . 131
5.10.2 Model . . . 132
5.10.3 Evaluation . . . 137
5.10.4 Link to Requirements . . . 138
5.11 Task Management . . . 139
5.11.1 Motivation . . . 139
5.11.2 Model . . . 140
5.11.3 Evaluation . . . 141
5.11.4 Link to Requirements . . . 141
5.12 Performance . . . 142
5.12.1 Motivation . . . 142
5.12.2 Background: DEVS Modelling and Simulation . . . 143
5.12.3 Model . . . 147
5.12.4 Evaluation . . . 150
5.12.5 Related Work . . . 153
5.12.6 Link to Requirements . . . 154

6 Modelverse as a Foundation for MPM **157**
6.1 Power Window Case Study . . . 157
6.1.1 Requirement 1: Domain-Specific Modelling . . . 158
6.1.2 Requirement 2: Activities . . . 159
6.1.3 Requirement 3: Process Modelling . . . 161
6.1.4 Requirement 4: Multi-User . . . 161
6.1.5 Requirement 5: Multi-Service . . . 162
6.1.6 Requirement 6: Multi-Interface . . . 162
6.1.7 Requirement 7: Model Sharing . . . 162

6.1.8 Requirement 8: Access Control . . . 163
6.1.9 Requirement 9: Megamodelling . . . 163
6.1.10 Requirement 10: Portability . . . 163
6.2 Live Modelling . . . 164
6.2.1 Motivation . . . 164
6.2.2 Background . . . 165
6.2.3 Running Examples . . . 168
6.2.4 Approach . . . 171
6.2.5 Evaluation . . . 182
6.2.6 Related Work . . . 186
6.3 Concrete Syntax . . . 188
6.3.1 Motivation . . . 189
6.3.2 Approach . . . 193
6.3.3 Evaluation . . . 197
6.3.4 Discussion . . . 202
6.3.5 Related Work . . . 203
6.4 Modelverse Debugging . . . 205
6.4.1 Motivation . . . 206
6.4.2 Background: DEVS Debugging . . . 206
6.4.3 Model . . . 217
6.4.4 Evaluation . . . 218
6.4.5 Related Work . . . 219

7 Conclusions **221**
7.1 Future Work . . . 225

A Modelverse State Specification **227**
A.1 Data representation . . . 227
A.2 CRUD interface . . . 228
A.2.1 Create . . . 228
A.2.2 Read . . . 229
A.2.3 Update . . . 232
A.2.4 Delete . . . 232

B Action Language Specification **235**
B.1 Documentation . . . 235
B.1.1 If condition . . . 235
B.1.2 While loop . . . 239
B.1.3 Break . . . 239
B.1.4 Continue . . . 239
B.1.5 Access . . . 241
B.1.6 Resolve . . . 242
B.1.7 Assign . . . 243
B.1.8 Function Call . . . 248
B.1.9 Return . . . 249
B.1.10 Constant . . . 249
B.1.11 Declare . . . 249
B.1.12 I/O . . . 252

B.1.13 Control Instructions 252
B.2 Primitives 256

Acknowledgments

I would like to thank a number of people, without whose help this thesis would not have been possible.

First off, this thesis would not be here, and I probably would not have started a scientific career, if it were not for my supervisor, Hans Vangheluwe. Already from my Bachelor thesis, your passionate encouragement has stimulated me to get the best out of myself. During my Masters, you introduced me to research and the scientific community, creating various opportunities for me to continue growing. Then during my PhD, you offered me the necessary freedom to explore and develop my interests, while offering guidance where necessary. Throughout these years I have learned a great deal from our discussions, both on and off topic.

Thanks to my colleagues, not only for the many collaborations and new insights, but also for the many lively discussions and lunches. In particular, thank you Simon for sharing an office with me and for the many ideas that started out on the whiteboard as mere scribbles, only to mature up to the point that they are included in this thesis. Thank you Bart, Claudio, István, Joachim, Ken, as well as the other members of the Ansymo research group, for the stimulating discussions and the cross-fertilizations with your respective domains. To Bentley, Maris, Sadaf, and Levi: thank you for hosting me when I visited MSDL at McGill.

Thanks to the members of my jury: Juan de Lara, Manuel Wimmer, Serge Demeyer, and Joachim Denil. Thank you for listening to this story and for your remarks and further insights that helped this thesis to reach its current level. They are much appreciated.

I would also like to thank the Research Foundation - Flanders (FWO) for providing me with the means to accomplish this work through my scholarship as Aspirant.

Thanks to my family and friends, who took my mind off things when I needed it and for helping me wherever possible. Last but not least, thank you Lieselotte. Thank you for your love and for understanding how important my work is to me.

Yentl Van Tendeloo
29 August 2018

Abstract

The complexity of engineered systems is rapidly increasing, mainly due to their heterogeneity at run time and design time. At run time, software controls hardware components in a feedback loop, the complete system has to interact (safely) with the environment, and often multiple such systems are connected over a network and have to cooperate to achieve a task. At design time, these run-time requirements often require multiple languages and tools to be combined, in order to create a single big system. With the advent of Cyber-Physical Systems and smart mechatronic systems of the Industry 4.0 initiative, engineers are facing challenges of an unprecedented magnitude.

To successfully and efficiently tackle the complexity of engineered systems, modelling- and simulation-based techniques are increasingly applied in the flow of the engineering work. *Model-Driven Engineering (MDE)* regards models as first-class concepts: before realizing the system, the various aspects of the system are modelled, allowing for analysis, simulation, and verification. These models are created by domain experts, meaning that these models are domain-specific. Within MDE, *Multi-Paradigm Modelling (MPM)* actively promotes this specialization. MPM advocates explicitly modelling every relevant aspect of the system, using the most appropriate formalism(s), at the most appropriate level(s) of abstraction, while explicitly modelling the process.

Despite the proposed advantages of MPM, such as lowering the cognitive gap and repeatability, current tools support only a subset of MPM. For example, they support domain-specific formalisms, but not process model enactment. This is not surprising, as MPM relies on several distinct research domains, all of which have to be integrated. To date, no foundation for MPM exists, which is also usable for future applications in the context of MPM.

We address this problem in three steps, forming the three main contributions of this thesis, related to the construction of a foundation for MPM. Within each contribution, several orthogonal contributions were made in the domains considered, highlighting the relevance and applicability of our MPM framework.

Our first contribution consists of creating a specification for a foundation for MPM, where we explicitly list the requirements for MPM tool support. These requirements are based on the definition of MPM and include both the explicitly stated requirements (e.g., support for domain-specific formalisms) and those left more or less implicit (e.g., support for multiple users). A minimal, though representative, power window case study is used to make these requirements more concrete.

Our second contribution consists of implementing these requirements for a prototype tool,

which we term the *Modelverse*. Given our assumption that MPM is useful for the modelling of complex systems, we apply MPM to the construction of this prototype itself. This serves as a case study for the application of MPM, highlighting its advantages and thereby further building the case for MPM. An MPM tool consists of several components (e.g., conformance relation, model management operations), all of which we discuss in detail. For each component, we motivate the use of MPM, consider the most appropriate formalism, present the model, and evaluate the use of MPM. Most components proved to be easier and less verbose to model, due to the higher level of abstraction: we could focus on the "what", instead of the "how". Several components, however, proved exceptionally advantageous to model explicitly and are listed next. Thanks to the application of MPM, we overcame existing limitations, resulting in further contributions.

- By explicitly modelling the conformance relation, we can dynamically and simultaneously support multiple types of conformance. This addresses different non-interoperable implementations of conformance found in today's (meta-)modelling tools, which hindered the use of model repositories.

- By explicitly modelling the physical type model, we can implement low-level model operations in the linguistic dimension instead of the physical dimension. This abstracts the physical implementation, allowing for different physical implementations while maximally reusing model management operations.

- By explicitly modelling process model enactment semantics by mapping it to a Statecharts model, the non-trivial implementation of concurrency can be omitted. The process also becomes susceptible to analysis techniques that are applicable to Statecharts.

- By explicitly modelling service orchestration, we obtain an explicitly modelled interface for a black-box service. This combines the functionality of the black-box component with the interface of a white-box component. We also gain analyzability of the different activities and their interaction.

- By explicitly modelling the action language semantics through graph transformation rules, we can automatically generate documentation and an interpreter. This makes the interpreter trivially portable between different platforms, while guaranteeing the exact same semantics.

- By explicitly modelling and simulating the performance, we can deterministically and efficiently assess performance in a variety of (hypothetical) scenario's.

Our third contribution consists of evaluating the presented prototype tool for its support of MPM and further research on MPM. This shows the applicability of our prototype and highlights the many potential directions for future work that our work enables, truly turning it into a foundation for MPM. As for the use in an MPM context, we evaluate support for each of the requirements originally mentioned in the power window case study. As for further research, we consider three distinct dimensions, each focussing on a different type of user that we envision will use the Modelverse. Our contribution in each dimension required full support for MPM, as offered by our tool.

- Live modelling allows modellers to alter the design model during the execution of that very same model, with modifications having a direct influence on the running

execution. Current approaches to live modelling do exist, though are ad-hoc and highly specific to the language under study. We propose a generic process, applicable to many different types of modelling languages, which we evaluate for three representative languages.

- Concrete syntax encompasses the different ways in which a model is presented towards the user. Current approaches are mostly ad-hoc and have severe restrictions, such as a strong coupling to the front-end and only support for visual languages. We propose a generic process for the perceptualization and rendering of abstract syntax models, which is agnostic to the rendering format.

- Debugging the Modelverse is complex due to the different interacting users, the distinction between different services, and the use of several interacting programs. Current approaches are not up to debugging several interacting programs simultaneously. Additionally, non-determinism, as introduced by the network and the other shared resources, renders fault reproduction impossible. We propose to debug a DEVS model instead of the actual code, thereby raising the level of abstraction and gaining full control over time.

Given that all these extensions were supported by the Modelverse out-of-the-box, we consider that it is usable for future research as well.

Nederlandstalige Samenvatting

De complexiteit van door de mens ontwikkelde systemen stijgt snel, voornamelijk door de heterogeniteit tijdens hun uitvoering en ontwikkeling. Tijdens de uitvoering controleert de ontwikkelde software verschillende (teruggekoppelde) hardware componenten, moet het volledige systeem (veilig) interageren met de omgeving, en vaak zijn meerdere zulke systemen verbonden over een netwerk om zo coöperatief een taak te vervullen. Tijdens de ontwikkeling impliceren die vereisten vaak de nood aan een combinatie van meerdere talen en *tools*, voor het creëren van één enkel groot systeem. Sinds de doorbraak van cyber-fysische systemen en slimme mechatronische systemen uit het Industry 4.0 initiatief, staan ingenieurs voor uitdagingen van een niet eerder geziene grootte.

Om succesvol en efficient de complexiteit van zulke systemen aan te pakken, worden modelleer- en simulatie-gebaseerde technieken steeds vaker gebruikt tijdens de ontwikkeling. *Model-Driven Engineering* (MDE) beschouwt modellen als basisconcepten: alvorens het eigenlijke systeem te realiseren worden de verschillende aspecten van het systeem gemodelleerd, wat analyse, simulatie en verificatie mogelijk maakt. Deze modellen worden ontwikkeld door domein-experts, wat betekent dat deze modellen domein-specifiek zijn. Binnen MDE is *Multi-Paradigm Modelling* (MPM) een specializatie die dit principe actief promoot. MPM pleit voor het expliciet modelleren van elk relevant aspect van het systeem, gebruik makende van de meest geschikte formalismen, op de meest geschikte niveaus van abstractie, terwijl eveneens het proces expliciet gemodelleerd wordt.

Ondanks de verscheidene voordelen van MPM, zoals het verkleinen van de cognitieve kloof tussen het probleem en de oplossing, ondersteunen bestaande *tools* slechts een deel van MPM. Er bestaan bijvoorbeeld *tools* die domein-specifieke formalismen ondersteunen, maar geen ondersteuning bieden voor het process. Dit is niet verrassend, daar MPM steunt op verschillende orthogonale onderzoeksdomeinen, dewelke allemaal geïntegreerd dienen te worden. Tot op heden bestaat er geen onderbouw voor MPM, dewelke tevens gebruikt kan worden voor toekomstige toepassingen binnen de context van MPM.

We behandelen dit probleem in drie fasen, die eveneens de drie hoofdbijdragen vormen van deze thesis. Binnen elke bijdrage werden verschillende orthogonale bijdragen gemaakt binnen het beschouwde domein, wat verder de relevantie en toepasbaarheid van onze aanpak onderstreept.

Onze eerste bijdrage bestaat uit het definiëren van een specificatie voor een onderbouw

voor MPM, dewelke expliciet de vereisten voor MPM oplijst. Deze vereisten zijn gebaseerd op de definitie van MPM en includeren zowel de expliciet vermelde vereisten (bv. ondersteuning voor domein-specifieke formalismen) als ook de impliciet gelaten vereisten (bv., ondersteuning voor meerdere gebruikers). Een minimaal doch representatief autotechnologisch voorbeeld van een automatisch ruitbedieningssysteem is gebruikt om deze vereisten te concretiseren.

Onze tweede bijdrage bestaat uit het implementeren van deze vereisten in een prototype, hetwelk we *Modelverse* noemen. Gegeven onze assumptie dat MPM een zinvolle techniek is voor het modelleren van complexe systemen, passen we MPM toe voor de constructie van dit prototype zelf. Dit dient eveneens als een gevalsstudie voor de toepasbaarheid van MPM, wat het mogelijk maakt de voordelen beter in de verf te zetten en verdere precedenten te scheppen. Een MPM *tool* bestaat uit verschillende componenten, zoals model conformiteit en operaties voor model beheer, dewelke allemaal in detail besproken zullen worden. Voor elke component motiveren we het gebruik van MPM, daarbij bepalende wat het meest geschikte formalisme is, gevolgd door een uitvoerige discussie van het model en een evaluatie van het gebruik van MPM. De meerderheid hiervan bleek makkelijker en compacter om te modelleren vanwege het hogere niveau van abstractie: we konden focussen op het “wat”, in plaats van op het “hoe”.

Bovendien bleken verschillende componenten exceptioneel voordelig om expliciet te modelleren. Dankzij de toepassing van MPM werd het namelijk mogelijk om bestaande limitaties weg te werken, wat de basis was voor verdere bijdragen.

- Door het expliciet modelleren van de conformiteit tussen modellen, bieden we dynamische en simultane ondersteuning voor meerdere types van conformiteit. Dit unificeert de verschillende, niet-interoperabele conformiteits implementaties, die gevonden worden in hedendaagse (meta-)modelleer *tools*, wat een beperkende factor is voor model uitwisseling.

- Door het expliciet modelleren van het fysieke type model, kunnen model operaties op een laag niveau in de linguistische dimensie geïmplementeerd worden, in plaats van in de fysieke dimensie. Dit abstraheert de fysieke implementatie en maakt het mogelijk om verschillende zulke implementaties te ondersteunen, terwijl model beheer operaties maximaal hergebruikt kunnen worden.

- Door het expliciet modelleren van de uitvoeringssemantiek van een proces model via een vertaling naar Statecharts, kan de niet-triviale implementatie van concurrente processen vermeden worden. Het proces wordt tevens vatbaar voor bestaande analyse technieken van Statecharts.

- Door het expliciet modelleren van de orchestratie van verschillende diensten, wordt het mogelijk om expliciet gemodelleerde interfaces aan te bieden voor arbitraire *black-box* diensten. Dit combineert de functionaliteit van zulke *black-box* componenten met de interface van *white-box* componenten. Daarenboven verkrijgen we analyseerbaarheid van de verschillende activiteiten en hun interactie.

- Door het expliciet modelleren van de actietaal semantiek door het gebruik van graaf transformatie regels, wordt het automatisch genereren van documentatie en interpreteerder mogelijk. Deze wordt bijgevolg triviaal over te dragen naar andere platformen, terwijl de exacte semantiek gegarandeerd behouden blijft.

- Door het expliciet modelleren en simuleren van de efficientie, kunnen we op deterministische en efficiente wijze de performantie inschatten in een resem (hypothetische) scenario's.

Onze derde bijdrage bestaat uit het evaluaren van het eerder vermelde prototype naar zijn ondersteuning voor MPM en toekomstig onderzoek in MPM. Dit toont de toepasbaarheid van het prototype aan en onderstreept de vele potentiële richtingen voor verder onderzoek, die mogelijk gemaakt worden door deze onderbouw. Voor het gebruik binnen een MPM context evalueren we de ondersteuning voor elk van de voornoemde vereisten binnen de gevalsstudie van het automatisch ruitbedieningssysteem. Voor het gebruik binnen toekomstig onderzoek beschouwen we drie verschillende dimensies, die elk focussen op een ander type gebruiker van onze onderbouw. Onze bijdrage in elke dimensie vereist steeds volledige ondersteuning voor MPM.

- Live modelleren staat modelleerders toe om een ontwerpmodel onmiddellijk te wijzigen tijdens de uitvoering van dat eigenste model, waarbij de wijzigingen een onmiddellijke impact hebben op de reeds lopende uitvoering. Bestaande technieken voor live modelleren zijn vaak ad hoc en hoogst specifiek voor de beschouwde taal. We stellen een generiek proces voor dat toegepast wordt voor drie verschillende representatieve talen.

- Concrete syntax omvat de verschillende manieren waarop een model kan gepresenteerd worden jegens gebruikers. Bestaande technieken zijn ook nu weer ad hoc of hebben significante restricties, zoals een sterke koppeling met de gebruikersinterface of het enkel aanbieden van ondersteuning voor visuele talen. We stellen een generiek process voor voor het perceptualizeren en uit te tekenen van een abstract syntax model, agnostisch van het gekozen medium.

- Het debuggen van de Modelverse is complex vanwege de potentiële interactie tussen gebruikers, het onderscheid tussen verschillende diensten, en het gebruik van verschillende interagerende programma's. Bestaande technieken zijn ontoereikend voor het debuggen van zulke systemen. Bovendien is de uitvoering vaak niet-deterministisch vanwege interferentie (bv., netwerkcommunicatie of gedeelde platformen), wat foutreproductie bemoeilijkt. We stellen een generieke techniek voor voor het debuggen van een DEVS model in plaats van de eigenlijke code, waardoor het niveau van abstractie verhoogd kan worden en we volledige controle krijgen over de tijd tijdens executie.

Gegeven dat deze drie uitbreidingen ondersteund werden door de Modelverse zonder ingrijpende wijzigingen, concluderen we dat deze eveneens bruikbaar is voor verder onderzoek binnen het domein van MPM.

Publications

The following peer-reviewed publications that I co-authored were included (partially) in this thesis:

1. VAN TENDELOO, Y., AND VANGHELUWE, H. The modular architecture of the Python(P)DEVS simulation kernel. In *Proceedings of the 2014 Spring Simulation Multiconference - TMS/DEVS* (2014), pp. 387–392

   Yentl and Hans came up with the ideas, Yentl implemented the approach and wrote the paper, Hans reviewed the paper. Briefly summarized in Section 5.12.2.

2. VAN TENDELOO, Y., AND VANGHELUWE, H. Activity in PythonPDEVS. In *Proceedings of ACTIMS* (2014)

   Yentl and Hans came up with the ideas, Yentl implemented the approach and wrote the paper, Hans reviewed the paper. Briefly summarized in Section 5.7.

3. VAN MIERLO, S., VAN TENDELOO, Y., BARROCA, B., MUSTAFIZ, S., AND VANGHELUWE, H. Explicit modelling of a Paralell DEVS experimentation environment. In *Proceedings of the 2015 Spring Simulation Multiconference - TMS/DEVS* (2015), pp. 860–867

   Simon and Yentl came up with the ideas, implemented the approach, and wrote the paper. Bruno, Sadaf, and Hans reviewed the paper. Partially incorporated in Section 6.4.2.

4. VAN TENDELOO, Y., AND VANGHELUWE, H. PythonPDEVS: a distributed Parallel DEVS simulator. In *Proceedings of the 2015 Spring Simulation Multiconference - TMS/DEVS* (2015), pp. 844–851

   Yentl and Hans came up with the ideas, Yentl implemented the approach and wrote the paper, Hans reviewed the paper. Briefly summarized in Section 5.12.2.

5. VAN TENDELOO, Y. Foundations of a multi-paradigm modelling tool. In *MoDELS ACM Student Research Competition* (2015), pp. 52–57

   Yentl came up with the ideas and wrote the paper. Partially incorporated in Chapter 4.

6. CARDOEN, B., MANHAEVE, S., TUIJN, T., VAN TENDELOO, Y., VANMECHELEN, K., VANGHELUWE, H., AND BROECKHOVE, J. Performance analysis of a PDEVS simulator supporting multiple synchronization protocols. In *Proceedings of the 2016 Symposium on Theory of Modeling and Simulation - TMS/DEVS* (2016), pp. 614–621

Yentl, Kurt, Hans, and Jan came up with the ideas; Ben, Stijn, and Tim implemented the approach and elaborated the ideas; Yentl, Ben, Stijn, and Tim wrote the paper; Yentl, Kurt, and Jan reviewed the paper. Briefly summarized in Section 5.7.

7. VAN TENDELOO, Y., AND VANGHELUWE, H. An overview of PythonPDEVS. In *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications* (2016), pp. 59–66

   Yentl came up with the ideas and implemented the approach and wrote the paper, Hans reviewed the paper. Briefly summarized in Section 5.12.2.

8. VAN MIERLO, S., VAN TENDELOO, Y., MEYERS, B., EXELMANS, J., AND VANGHELUWE, H. SCCD: SCXML extended with Class Diagrams. In *Proceedings of the Workshop on Engineering Interactive Systems with SCXML* (2016), pp. 2:1–2:6

   Hans came up with the ideas; Joeri, Simon, and Yentl implemented the approach; Simon and Bart wrote the paper; Yentl and Hans reviewed the paper. Partially incorporated in Section 2.4.5.

9. VAN MIERLO, S., VAN TENDELOO, Y., MEYERS, B., AND VANGHELUWE H. *Domain-Specific Modelling for Human-Computer Interaction.* Springer, 2017, pp. 435–463

   Bart came up with the ideas; Yentl, Simon, and Bart implemented the examples and wrote the paper; Hans reviewed the paper. Incorporated in Section 2.1.

10. VAN MIERLO, S., VAN TENDELOO, Y., AND VANGHELUWE, H. Debugging Parallel DEVS. *SIMULATION 93*, 4 (2017), 285–306

    Simon and Yentl came up with the ideas, implemented the approach, and wrote the paper; Hans reviewed the paper. Partially incorporated in Section 6.4.2.

11. VAN TENDELOO, Y., AND VANGHELUWE, H. An evaluation of DEVS simulation tools. *SIMULATION 93*, 2 (2017), 103–121

    Yentl and Hans came up with the ideas; Yentl updated the evaluation criteria, implemented the examples, and wrote the paper; Hans reviewed the paper. Incorporated in Section 5.12.2.

12. CARDOEN, B., MANHAEVE, S., VAN TENDELOO, Y., AND BROECKHOVE, J. A PDEVS simulator supporting multiple synchronization protocols: implementation and performance analysis. *SIMULATION 93* (2017)

    Yentl and Jan came up with the ideas; Ben and Stijn implemented the approach and elaborated the ideas; Yentl, Ben, and Stijn wrote the paper; Yentl and Jan reviewed the paper. Briefly summarized in Section 5.7.

13. VAN TENDELOO, Y. VAN MIERLO, S., AND VANGHELUWE, H. Time- and space-conscious omniscient debugging of Parallel DEVS. In *Proceedings of the 2017 Spring Simulation Multiconference - TMS/DEVS* (2017), pp. 1001–1012

    Yentl came up with the ideas and implemented the approach; Yentl and Simon wrote the paper; Hans reviewed the paper. Partially incorporated in Section 6.4.2.

14. VAN TENDELOO, Y., AND VANGHELUWE, H. The Modelverse: a tool for multi-paradigm modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference* (2017), pp. 944–955

    Yentl came up with the ideas and implemented the approach and wrote the paper, Hans reviewed the paper. Partially incorporated in Section 6.1.

15. VAN TENDELOO, Y., AND VANGHELUWE, H. Classic DEVS modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference* (2017), pp. 644–656

    Yentl came up with the ideas and implemented the approach and wrote the paper, Hans reviewed the paper. Partially incorporated in Section 2.4.6.

16. VAN TENDELOO, Y., AND VANGHELUWE, H. Increasing performance of a DEVS simulator by means of computational resource usage "activity" models. *SIMULATION 93*, 12 (2017), 1045–1061

    Yentl came up with the ideas and implemented the approach and wrote the paper, Hans reviewed the paper. Briefly summarized in Section 5.7.

17. VAN TENDELOO, Y., AND VANGHELUWE, H. Explicitly modelling the type/instance relation. In *Proceedings of MODELS 2017 Satellite Event* (2017), pp. 392–398

    Yentl came up with the ideas and implemented the approach and wrote the paper, Hans reviewed the paper. Incorporated in Section 5.6.

18. VAN TENDELOO, Y., VAN MIERLO, S., MEYERS, B., AND VANGHELUWE, H. Concrete syntax: a multi-paradigm modelling approach. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (2017), pp. 182–193

    Yentl and Simon came up with the ideas; Yentl, Simon, and Bart elaborated on the idea; Yentl implemented the approach; Yentl and Simon wrote the paper; Bart and Hans reviewed the paper. Incorporated in Section 6.3.

19. VAN TENDELOO, Y., AND VANGHELUWE, H. Extending the DEVS formalism with initialization information. ArXiv e-prints (2018)

    Yentl and Hans came up with the ideas; Yentl implemented the approach and wrote the paper; Hans reviewed the paper. Partially incorporated in Section 2.4.6.

20. VAN MIERLO, S., VAN TENDELOO, Y., DÁVID, I., MEYERS, B., GEBREMICHAEL, A., AND VANGHELUWE, H. A multi-paradigm approach for modelling service interactions in model-driven engineering processes. In *Proceedings of the 2018 Spring Simulation Multiconference - Mod4Sim* (2018), pp. 565–576

    Bart and Hans came up with the ideas; Yentl and Addis elaborated on the idea; Addis and Yentl implemented the approach; Simon, Yentl, and István wrote the paper; Hans and Addis reviewed the paper. Incorporated in Section 5.8 and Section 5.9.

21. VAN TENDELOO, Y., AND VANGHELUWE, H. Introduction to Parallel DEVS modelling and simulation. In *Proceedings of the 2018 Spring Simulation Multiconference - Mod4Sim* (2018), pp. 613–624

Yentl and Hans came up with the ideas; Yentl implemented the approach and wrote the paper; Hans reviewed the paper. Partially incorporated in Section 2.4.6.

22. VAN TENDELOO, Y., AND VANGHELUWE, H. Unifying Screen- and Modelsharing. In *Proceedings of the 2018 WETICE Conference* (2018), pp. 127–132

    Yentl came up with the ideas, implemented the approach, and wrote the paper; Hans reviewed the paper. Partially incorporated in Section 6.3.

23. VAN TENDELOO, Y., AND VANGHELUWE, H. DEVS Modelling and Simulation of a Multi-Paradigm Modelling Tool. In *Proceedings of the 2018 Summer Simulation Multiconference - SCSC* (2018), pp. 27–38

    Yentl came up with the ideas, implemented the approach, and wrote the paper; Hans reviewed the paper. Incorporated in Section 6.4 and Section 5.12.

The following peer-reviewed publications that I co-authored were not included in this thesis:

1. VAN TENDELOO, Y., AND VANGHELUWE, H. Logisim to DEVS translation. In *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications* (2013), pp. 13–20

   Hans came up with the ideas; Yentl implemented the approach; Yentl and Hans wrote the paper.

2. VAN TENDELOO, Y., AND VANGHELUWE, H. Teaching the fundamentals of the modelling of cyber-physical systems. In *Proceedings of the 2016 Spring Simulation Multiconference - TMS/DEVS* (2016), pp. 646–653

   Hans and Yentl came up with the ideas and wrote the paper.

3. GOMES, C., VAN TENDELOO, Y., DENIL, J., DE MEULENAERE, P., AND VANGHELUWE, H. Hybrid system modelling and simulation with Dirac deltas. In *Proceedings of the 2017 Spring Simulation Multiconference - TMS/DEVS* (2017), pp. 1049–1060

   Claudio came up with the idea; Yentl, Joachim, and Claudio elaborated on the idea; Claudio implemented the approach and wrote the paper; Yentl, Joachim, Paul, and Hans reviewed the paper.

4. DÁVID, I., MEYERS, B., VANHERPEN, K. VAN TENDELOO, Y., BERX, K., AND VANGHELUWE, H. Modeling and enactment support for managing inconsistencies in heterogeneous systems engineering processes. In *Proceedings of MODELS 2017 Satellite Event* (2017), pp. 145–154

   István, Bart, Ken, and Yentl came up with the idea; István, Ken, Bart, and Kristof elaborated on the ideas; István wrote the paper; Bart, Ken, Yentl, Kristof, and Hans reviewed the paper.

5. DÁVID, I., VAN TENDELOO, Y., AND VANGHELUWE, H. Translating Engineering Workflow Models to DEVS for Performance Evaluation. In *Proceedings of the 2018 Winter Simulation Conference* (2018) (accepted)

   István, Yentl, and Hans came up with the idea; István and Yentl elaborated on the ideas, implemented the approach, and wrote the paper; Hans reviewed the paper.

6. VAN TENDELOO, Y., AND VANGHELUWE, H. Classic DEVS Modelling and Simulation. In *Proceedings of the 2018 Winter Simulation Conference* (2018) (accepted)

   Updated version of our previous paper at the 2017 Winter Simulation Conference.

Overview of Activities

During my PhD, I participated in a number of scientific activities that were (to some extent) related to my research. A (non-exhaustive) list is included here.

Teaching

- Lab Sessions "Modelling of Software-Intensive Systems" course at University of Antwerp (2014 - 2018).
- Several Lectures "Modelling of Software-Intensive Systems" course at University of Antwerp (2014 - 2018).
- Several Lectures "Computer-Systems and Architecture" course at University of Antwerp (2014 - 2018).
- Lab Sessions of the 5th International Summer School on Domain-Specific Modelling (DSM-TP 2015).
- Lab Sessions of the 6th International Summer School on Domain-Specific Modelling (DSM-TP 2016).
- Tutorial "DEVS Modelling and Simulation" at SpringSim'16, 3 April 2016.
- Tutorial "Introduction to Parallel DEVS Modeling and Simulation" at SpringSim'17, 23 April 2017.
- Tutorial "Classic DEVS Modelling and Simulation" at WinterSim'17, 6 December 2017.
- Tutorial "Introduction to Classic DEVS Modelling and Simulation" at SpringSim'18, 15 April 2018.
- Tutorial "Practical Classic DEVS Modelling and Simulation" at SpringSim'18, 15 April 2018.
- Tutorial "Introduction to Classic DEVS" at SummerSim'18, 9 July 2018.
- Tutorial "An Introduction to Statecharts Modeling and Simulation" at SummerSim'18, 9 July 2018.

Participation in Scientific Activities

- International Symposium on Distributed Simulation and Real Time Applications 2013 (DS-RT'13) (Delft, the Netherlands; 30 October - 1 November 2013)
- Domain Specific Modelling - Theory and Practice 2014 (DSM-TP'14) (Antwerp, Belgium; 25-29 August 2014)
- Spring Simulation Multi-Conference 2015 (SpringSim'15) (Alexandria, VA, USA; 12 - 15 April 2015)
- Domain Specific Modelling - Theory and Practice 2015 (DSM-TP'15) (Antwerp, Belgium; 24-28 August 2015)
- International Conference on Model Driven Engineering Languages and Systems 2015 (MoDELS'15) (Ottawa, Canada; 27 September - 2 October 2015)
- Spring Simulation Multi-Conference 2016 (SpringSim'16) (Pasadena, CA, USA; 3 - 6 April 2016)
- Workshop Les Journées DEVS francophones: Théorie et Applications (JDF'16) (Cargèse, France; 11 - 15 April 2016)
- CAMPaM Workshop 2016 (Bellairs, Barbados; 29 April - 6 May 2016)
- Spring Simulation Multi-Conference 2017 (SpringSim'17) (Virginia Beach, VA, USA; 23 - 26 April 2017)
- International Conference on Model Driven Engineering Languages and Systems 2017 (MoDELS'17) (Austin, TX, USA; 17 - 22 September 2017)
- International Conference on Software Language Engineering 2017 (SLE'17) (Vancouver, Canada; 23 - 24 October 2017)
- Winter Simulation Conference 2017 (WSC'17) (Las Vegas, NV, USA; 3 - 6 December 2017)
- Master Class: From Products to Product Families: Leverage your Product Portfolio (Antwerp, Belgium; 1 February 2018)
- Spring Simulation Multi-Conference 2018 (SpringSim'18) (Baltimore, MD, USA; 15 - 18 April 2018)
- International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises 2018 (WETICE 2018) (Paris, France; 27 - 29 June 2018)
- Summer Simulation Multi-Conference 2018 (SummerSim'18) (Bordeaux, France; 9 - 12 July 2018)

Reviewing

- Workshop on Activity in Modelling and Simulation (ACTIMS)
- Spring Simulation Multi-Conference (SpringSim)
- Winter Simulation Multi-Conference (WSC)
- SIMULATION: Transactions of the Society for Modeling Simulation International (SIMULATION)
- ACM Transaction on Modeling and Computer Simulation (TOMACS)
- Computer Languages, Systems and Structures (COMLAN)
- The Handbook of Formal Methods in Human-Computer Interaction (FoMHCI)
- Simulation Modelling Practice and Theory (SimPaT)

List of Figures

2.1 Terminology 11
2.2 Abstract syntax definition for the nuclear power plant domain (some subclasses omitted) 12
2.3 Concrete syntax definition for the nuclear power plant domain (excerpt) . 13
2.4 Screenshot of an example nuclear power plant instance 14
2.5 Statechart equivalent to the behaviour shown in Figure 2.4 16
2.6 Top-level transformation schedule 18
2.7 Transformation schedule for valves 18
2.8 Example transformation rule for translational semantics 18
2.9 Example transformation rule for operational semantics 19
2.10 Example FSA of a home security alarm system. 20
2.11 Abstract Syntax of Finite State Automata. Runtime-only concepts are shown in bold. The association "Transition" has additional attributes defined on it. 21
2.12 Abstract Syntax of Discrete Time Causal Block Diagrams. Runtime-only concepts are shown in bold. 22
2.13 Example DTCBD. 22
2.14 Abstract Syntax of Continuous Time Causal Block Diagrams. 23
2.15 Example CTCBD. 23
2.16 Expand rule for an integrator block. 24
2.17 Optimize rule for an invertor block. 24
2.18 Example Petri Net with reachability graph. 25
2.19 Bouncing ball example in SCCD (abstracted). 27
2.20 Simulation trace with hypothetical negative simulation time (grayed out). 32
2.21 Various options for shifting the police model. 32
2.22 Example FTG+PM for safety analysis of a simplified power window. . . . 33

3.1 Meaning of a simple Petri Nets metamodel. U represents all possible instances, including those violating the structure; T_S represents all instances that conform structurally, but not necessarily to all the constraints; $T_{S \wedge C}$ represents all instances that conform both structurally and fulfil all constraints. 36
3.2 Four-layered architecture as popularized by the OMG. 37
3.3 Three different classification dimensions: ontological, linguistcal, and physical. All dimensions are orthogonal to each other. 38
3.4 Multi-level modelling. 39

4.1 Three types of users in relation to the meta-hierarchy. 47
4.2 The relation between the model and the system. [337] 48

5.1 Overview of all components of the Modelverse, together with the section in which their model is described. 68
5.2 SCCD model of the GUI (excerpt). 71
5.3 Screenshot of the GUI with a loaded CBD model. 72
5.4 Abstract version of the client wrapper Statechart. 75
5.5 Network protocol SCCD model at server side (abstracted). 78
5.6 Network protocol SCCD model at client side (abstracted). 79
5.7 Initial bootstrap FTG including metadata and access control (excerpt). Instance-of links are parameterized with a conformance semantics (e.g., *conformance_mv*) and type mapping model (e.g., *0*). 87
5.8 First example language: use of containment links. 91
5.9 Second example language: use of node multiplicities. 92
5.10 Third example language: use of multiple inheritance. 92
5.11 AToMPM's M_3^{AS}, taken directly from AToMPM itself. 94
5.12 Modelverse's M_3^{AS}. 95
5.13 Overview of languages, models, and relations in the Modelverse. 97
5.14 An algorithm spanning multiple layers in the linguistic dimension, violating strict metamodelling. 101
5.15 $LTM_{\perp}$, allowing for any element to connect to any other element. 102
5.16 $LTM_{\perp}$ added in the linguistic dimension, which is identical to the one in the physical dimension. 103
5.17 Different modelling hierarchies for the model *my_PN*, as seen through two different linguistic views. 104
5.18 Changing the physical metamodel with something else, as long as there is still a mapping to $LTM_{\perp}$. SQL metamodel not expanded due to space constraints. 105
5.19 Two different ontological views on the same model. The elements accessed by the algorithm are shown in light blue. Only $conformance_{\perp}$ complies with strict metamodelling. 106
5.20 Overview of the complete procedure: (1) reinterpret the model as instance of $LTM_{\perp}$, (2) execute the algorithm on the graph representation, (3) reinterpret the model again using the initial metamodel. All steps happen on the background and the user only sees the composite operation. 107
5.21 Snapshot of air traffic simulation load distribution at start of simulation. . 111
5.22 Snapshot of air traffic simulation load distribution during simulation. . . . 111
5.23 Snapshot of air traffic simulation load distribution after load balancing. . . 111
5.24 Activity as an optional extension to both the model and simulator. 112
5.25 Simplified city lay-out model. 113
5.26 Execution time results for the city layout model. 113
5.27 Results of using the polymorphic scheduler data structure, normalized for the polymorphic scheduler. 115
5.28 Results of using the polymorphic scheduler data structure with different phases. 115
5.29 Performance of conservative and optimistic synchronization for the modified PHold model. 116

5.30 Process model of the example. 120
5.31 Automatic activity: protocol implemented to communicate with an external service. 121
5.32 Problematic process model with a naive mapping to SCCD constructs. . . 126
5.33 Optimize rules. 127
5.34 Orchestrator rule. 127
5.35 Activity rule. 127
5.36 Fork rule. 127
5.37 Join rule. 127
5.38 Decision rule. 127
5.39 Power window FTG+PM. 129
5.40 Power window FTG+PM from Figure 5.39 mapped to SCCD. 129
5.41 Abstract syntax of the Action Language. Associations whose name are within square brackets indicate optional associations. All associations have a maximum cardinality of 1. 133
5.42 Abstract syntax of a Fibonacci algorithm written in the Action Language. 134
5.43 Graph transformation rule for the *If* construct to switch to the *else* block. . 136
5.44 Example optimization rule for the Action Language: constant folding. . . 138
5.45 Task manager SCCD model (abstracted). 140
5.46 "Traffic" model, shown for 2 segments. 146
5.47 Benchmark results for the "Traffic" benchmark. The left figure uses a logarithmic scale, whereas the right figure is zoomed in on the fastest tools and uses a linear scale. 147
5.48 Distribution of time taken for rule generation in the MvK. 149
5.49 Influence of MvI - MvK latency. 151
5.50 Influence of MvK - MvS latency. 151
5.51 Actual execution results. 152
5.52 Varying MvI - MvK latency. 153
5.53 Varying MvK - MvS latency. 153

6.1 FTG+PM of our example: the development and verification of a simplified power window. 158
6.2 Meta-modelling hierarchy of the example. 159
6.3 Plant metamodel, describing allowed constructs for a plant model. 159
6.4 Environment model. 159
6.5 Safety query model. 159
6.6 Control model. 159
6.7 Plant model. 160
6.8 Architecture model. 160
6.9 Rule for transforming the environment model to an encapsulated Petri Net. 160
6.10 Combined Petri Net, automatically generated from the DSL models. . . . 161
6.11 State of the game before and after decreasing the jump height parameter. . 166
6.12 Example FSA of a home security alarm system. 169
6.13 Example DTCBD. 170
6.14 Example CTCBD. 170
6.15 Diagrammatic overview of live programming. Full lines represent operations, dotted lines represent typing relations. 172

6.16 The partial runtime model of the example CTCBD, as an instance of the DTCBD language. . . . 173
6.17 The full runtime models of the examples. . . . 174
6.18 Sanitization in FSAs. . . . 177
6.19 Sanitization in DTCBDs. . . . 178
6.20 New design model for CTCBDs. . . . 179
6.21 Overview of our approach applied to an FSA mode, including traceability links. . . . 180
6.22 Overview of our approach applied to a DTCBD model, including traceability links. . . . 180
6.23 Overview of our approach applied to a CTCBD model. Only some interesting traceability links are shown. . . . 181
6.24 Overview of our approach, as an FTG+PM model. . . . 181
6.25 Live modelling for FSAs, before change. . . . 184
6.26 Live modelling for FSAs, after removing current state and setting new initial state. . . . 184
6.27 Live modelling for DTCBDs, before change. . . . 185
6.28 Live modelling for DTCBDs, after adding and connecting the multiplication block. . . . 185
6.29 Implementation of live modelling for CTCBDs. . . . 185
6.30 Simulation trace, where the constant "g" is changed at around time 63. . . . 186
6.31 Plotted trace of the CBD model, with $k = 1$, $m = 1kg$, $y_0 = 20cm$, $v_0 = 1\frac{cm}{s}$, and $g = 10\frac{cm}{s^2}$. . . . 190
6.32 Graphical representation of the trace in Figure 6.31. . . . 191
6.33 Using different set of icons. . . . 192
6.34 Circle lay-out of the same model. . . . 192
6.35 Alternative representation using a more complex mapping. . . . 192
6.36 Overview of the approach. . . . 193
6.37 Overview of the approach with an example for CBDs. . . . 194
6.38 MM_{render} for graphical visualization. . . . 194
6.39 Example rule for CBD perceptualization. . . . 195
6.40 Approach with multiple GUI front-ends. . . . 198
6.41 Approach with multiple *MM$_{Render}$* models. . . . 199
6.42 Approach with multiple mappers to the same *MM$_{Render}$*. The same tool is used for both models, though different instances. . . . 199
6.43 Concrete Syntax definition to automatically generate the perceptualization and comprehension operations. . . . 203
6.44 Simulation time and steps. . . . 207
6.45 PythonPDEVS Statechart. . . . 209
6.46 PythonPDEVS Statechart augmented with debugging functionality. . . . 210
6.47 Overhead of omniscient debugging in forward simulation. . . . 211
6.48 Overview of periodic state saving approach. Green states (light) are stored, red (dark) states are not. Yellow lines indicate a point at which a snapshot is made. . . . 212
6.49 Overhead of omniscient debugging in function of state size (logaritmic scale). 213
6.50 Jump latency for copy and periodic state saving. . . . 213
6.51 Memory use of copy state saving. . . . 214
6.52 Memory use of periodic state saving. . . . 214

B.1 *If* construct needs to evaluate the condition. 236
B.2 *If* construct needs to evaluate the then branch. 236
B.3 *If* construct needs to evaluate the else branch, and there is one. 237
B.4 *If* construct needs to evaluate the else branch, but there is none. 237
B.5 *While* construct needs to evaluate the condition. 238
B.6 *While* construct must loop. 238
B.7 *While* construct must terminate. 239
B.8 *Break* construct. 240
B.9 *Continue* construct. 240
B.10 *Access* construct must fetch the referred value. 241
B.11 *Access* construct needs to access the value. 241
B.12 *Resolve* construct resolves a non-global element. 242
B.13 *Resolve* construct resolves a global element. 242
B.14 *Assign* construct reads out the symbol to assign to. 243
B.15 *Assign* construct reads out the value to assign. 243
B.16 *Assign* construct performs the actual assigment. 244
B.17 *Call* construct resolves function with no parameters. 244
B.18 *Call* construct resolves function with parameters. 245
B.19 *Call* construct invokes with no parameters. 245
B.20 *Call* construct invokes with parameters. 246
B.21 *Call* construct evaluates first of multiple parameters. 246
B.22 *Call* construct evaluates first and only parameter. 247
B.23 *Call* construct evaluates last of multiple parameters. 247
B.24 *Call* construct evaluates next of multiple parameters. 248
B.25 *Return* construct returns with no returnvalue. 249
B.26 *Return* construct evaluates the returnvalue. 250
B.27 *Return* construct returns the evaluated returnvalue. 250
B.28 *Constant* construct. 251
B.29 *Declare* construct for a local variable. 251
B.30 *Global* construct for a global variable. 252
B.31 *Input* construct for fetching external input. 253
B.32 *Output* construct must evaluate output value. 253
B.33 *Output* construct must output the evaluated value. 254
B.34 Instruction has finished execution and has a next link. 254
B.35 Instruction has finished execution but has no next link. 255

List of Tables

3.1 Tool comparison as to which domains they support. 44

4.1 Modelling operations in the MvK. 60
4.2 Megamodelling operations in the MvK. 61
4.3 Activity operations in the MvK. 61
4.4 Process operations in the MvK. 61
4.5 Access control operations in the MvK. 62
4.6 Service operations in the MvK. 62
4.7 Create operations in the MvS. 64
4.8 Read operations in the MvS. 64
4.9 Delete operations in the MvS. 65

5.1 Complexity of different scheduler types. k is the number of imminent models and n is the total number of models in the simulation. 114
5.2 Summary of related work. (● - Supports, ◐ - Partially supports, ○ - Does not support) 123
5.3 General evaluation, based on [208, 278]. 145
5.4 DEVS-specific evaluation, based on [179, 313, 317]. 146
5.5 MvS operations and the measured calibration results. 149

6.1 A comparison with several other DEVS debugging tools. 216

B.1 Primitive functions modifying primitive datavalues. If a Value is taken or returned, this refers to the value of the returned node. 257
B.2 Lower-level primitive functions to implement. If a Value is taken or returned, this refers to the value of the returned node. 258

Chapter 1

Introduction

The complexity of engineered systems is rapidly increasing, mainly due to their heterogeneity at run time and design time. At run time, software controls hardware components in a feedback loop, the complete system has to interact (safely) with the environment, and often multiple such systems are connected over a network and have to cooperate to achieve a task. At design time, these runtime requirements often require multiple languages and tools to be combined, in order to create a single big system. With the advent of Cyber-Physical Systems (CPS), smart mechatronic systems of the Industry 4.0 initiative, and the Internet-of-Things (IoT), engineers are facing challenges of an unprecedented magnitude.

To successfully and efficiently tackle the complexity of engineered systems, modelling- and simulation-based techniques are increasingly applied in the flow of the engineering work. *Model-Driven Engineering (MDE)* [157] regards models as first-class concepts: before realizing the system, stakeholders (*e.g.*, control engineers, application engineers, test engineers) build models of the various aspects (*e.g.*, physical, mechanical, software) of the system, resulting in a virtual product which can be analysed, simulated, and verified. Stakeholder models capture the parts of the virtual product relevant to the stakeholder [52]. Often, stakeholders specialize in different domains and, therefore, their models are domain-specific. Within MDE, *Multi-Paradigm Modelling (MPM)* [200] actively promotes this specialization. MPM advocates modelling every relevant aspect of the system explicitly, using the most appropriate formalism(s), at the most appropriate level(s) of abstraction, while explicitly modelling the process.

Despite the proposed advantages of MPM for currently engineered systems, tool support for MPM remains rather limited. While some tools exist, they often focus exclusively on a selection of aspects of MPM, such as language engineering or process modelling. This is not surprising, as each of these aspects constitutes its own research domain, often with its own community. And while some academic tools exist that combine several such aspects, industrial support is restricted to a single domain. Existing tools are often constructed in an ad-hoc way, making future extensions difficult. To date, no unified foundation for MPM exists that incorporates all aspects of MPM, while also being usable for future extensions and applications in the domain of MPM.

In this thesis, we address this problem in three steps. First, we create a specification for a

Multi-Paradigm Modelling foundation. This specification is based on the tool requirements defined by MPM, both explicitly and implicitly, as well as on a representative MPM case study. Second, we construct a tool implementing this specification, using Multi-Paradigm Modelling techniques. The application of MPM is expected to aid the construction of the tool, thereby also building the case for MPM. Third, we consider several research directions in the domain of MPM, illustrating how contributions in this direction can build on our MPM tool. This shows the applicability of our prototype and highlights the many potential directions for future work that our work enables. In summary, this thesis presents a foundation for MPM together with a prototype implementation, built following an MPM approach, and applied in several research directions.

1.1 Motivation

We will now elaborate on the motivation for each of these three aspects of this thesis. This thesis explicitly builds on the assumption that MPM is indeed a useful approach, which we establish first.

1.1.1 Usefulness of MPM

The assumption that MPM is useful, is based on various pieces of empirical evidence, gathered throughout the years in both academics and industry. Most experiences are however related to MDE, instead of MPM, as they don't always use the *most* appropriate formalism for the domain (e.g., a general purpose formalism) or don't model everything (e.g., not the process). Nonetheless, most applications use at least some formalisms that are more appropriate than a general purpose programming language, and are therefore relevant in this context. In the end, MPM is a specialization of MDE, meaning that all advantages of MDE are also applicable for MPM.

MPM has been successfully applied by several companies. At ASML, the implementation of servo controllers for lithoscanners only takes a couple of minutes, instead of multiple days, as they can automatically generate code from their models, which had to be created anyway for reasons of analysis and documentation [252]. At Ericsson, the implementation of the channel estimation of the LTE-A uplink test bed of 4G telecommunication systems using models was considered a success [56], though no hard numbers were mentioned. At Motorola, consistent benefits from MDE and code generation were noticed, such as a $1.2\times$ to $4\times$ overall reduction in defects, a $3\times$ improvement in phase containment of defects, and a $2\times$ to $8\times$ productivity improvement [34]. At Panasonic, initial results are also satisfying [242]. General Motors, and in the automobile indistry in general, embrace Model-Based System Engineering (MBSE) for modelling, simulation, testing, and analysis [240].

Several applications were also made in an academic context. A gesture-based 3D application was created in merely three days, despite the complex requirements [94]. The integration of a quantitative analysis model went from several days to a few hours through the use of an Model-Based Engineering (MBE) platform [47]. Embedded control software for mechatronic systems could be developed and model checked within 12 days, which would normally take at least six weeks [129]. Supervisory controllers were modelled and model checked for an industrial-size case study [253]. Other applications are in the domain of

reconfigurable mechatronic systems [140], software architectures [119], model-integrated computing [275], and tool interoperability [104].

MPM is never truely necessary, as it is always possible to hand-code everything [255]. Claiming that MPM is therefore useless is naive, as the same could then be said of high-level programming languages, since all code could be written in assembly just as well. Reported advantages of using MPM are decreased development time [34, 125, 200], faster code execution [34], reduced costs [125], analyzability [145, 200], better documentation [200, 255, 341], or less errors [34, 125, 185, 341]. It is therefore natural that there is raising interest in MPM in research [321] and in teaching [291, 292, 311].

Given this body of evidence, we assume MPM to be useful throughout the remainder of this thesis. Note, however, that sometimes negative speedups were noted for some components [34]. Thus sometimes code might still be required, though this is not disallowed by MPM: code might very well be the most appropriate formalism.

Multi-Paradigm Modelling builds on top of some enabling techniques [316], each having their own research community and motivating examples. We next look at some of the most prominent techniques individually (details in Chapter 2), referring to applications and experiences in this research domain. As MPM builds on top of these techniques, experiences with them might be applicable to MPM as well.

Language Engineering

Thanks to language engineering, dedicated domain-specific languages can be created for each individual aspect of the system. Instead of using some general-purpose (modelling) language, domain-specific languages are often claimed to be less verbose and more intuitive to domain-experts. The usefulness of language engineering has been shown on several occasions [281]. Programming languages are simply not well-suited for some domains, such as concurrency: threads thoroughly disrupt the essential determinism of languages [176].

Activities

Thanks to activities, (domain-specific) models can be used for more than documentation only. Instead of only having syntax, activities make it possible to describe domain-specific semantics as well. Examples are simulation, optimization, execution, code generation, coupling with external tools, and so on. While we leave open how these activities are defined, model transformations are popular in the MDE community, as it relies on (often visual) pattern matching and rewriting. This is based on graph transformations, which has an extensive research community, but which we will not delve into in this thesis. Activities, and in particular model transformations, are essential to support MDE [42, 257]. Indeed, without activities, these domain-specific models are merely documentation, and while useful, this does not unleash the full potential of MDE.

Process Modelling

Process modelling has become necessary due to the rising complexities of the design process. By explicitly modelling the process, and potentially enacting it afterwards, the process becomes structured and deterministic, thereby avoiding mistakes. With process enactment,

the process is executed: first the initial activity is executed on the defined model artefacts, thereby generating or updating model artefacts, and then it continues on to the next activities. Processes can support concurrency, thereby allowing multiple modellers to work on different (manual) activities simultaneously. Automated activities can similarly happen concurrently, and possibly even during the manual operations. Furthermore, process modelling is a first step towards process optimization. The usefulness of process modelling has been shown on several occasions [93, 184], and is especially important during concurrent design, for example for embedded control software [129].

Summary

The literature provides various pieces of evidence suggesting the usefulness of MPM and the various enabling technologies it encompasses, both in academics and industry. Building on this information, this thesis assumes that MPM has indeed proven its usefulness.

1.1.2 Tool Support

The many applications of MPM already touched upon some of the requirements for a full-fledged MPM tool. None of the currently existing tools, however, comes close to fulfilling all the posed requirements. This is understandable, as most tools specialize in a specific research domain. Only some of these tools were designed with MPM in mind, and even those that focus on MPM, only implemented several aspects.

The literature on MPM also indicates that current tool support is either missing, rare [277], or sub-optimal [125, 141, 147, 184, 242, 290]. Nonetheless, tool support is deemed essential for the further development and application of the domain [47, 93, 141, 147, 185, 242, 277, 290]. With our proposed foundation and accompanying tool, we hope to unlock the full potential of MPM, thereby enabling future research in MPM and its adoption.

1.1.3 Use of MPM Techniques

A tool that fully supports all aspects of MPM is almost certain to be a complex piece of software. Not only will it have to cope with all aspects of language engineering, activities, and processes, but it should also run as a service for both computation (i.e., run activities "in the cloud") and storage (i.e., a (meta)model repository), while interacting with multiple geographically-distributed users with different backgrounds. These tool requirements, further explored in chapter 4, largely overlap with the scenario for which MPM was conceived. Although there is no physical aspect to an MPM tool, it consists of complex pieces of software (i.e., the different clients and the server) serving geographically distributed users interacting over a network (i.e., to allow for use as a service) and communicating with an environment (i.e., the end-user) in a performance-constrained way (i.e., have usable performance). Given our previous assumption that MPM is useful in such circumstances, we intend to apply MPM to the development of the MPM tool.

When MPM techniques are applied in this thesis, we link back to the original requirements, mentioning how the use of MPM has aided us in achieving them. As will be shown, each aspect of the tool has a well-founded reason for the use of modelling in that specific formalism, and is not merely modelled "for the sake of modelling". The use of MPM techniques does not necessarily enable new things to be developed, but will generally make

its development more efficient. This can be compared to programming in either assembly or Python: both languages can be used to create most software, though development in Python will often be faster due to the lower cognitive gap and decreased verbosity. By applying MPM to our prototype tool itself, we further build the case for MPM, while also showing our tool to be sufficiently expressive.

In the literature, several approaches partially adopted an MPM approach by modelling some aspects explicitly, resulting in advances in their respective fields. Examples include an explicit model weaving model to be used as a specification [147], explicit traceability models [277], explicit dependency model in the process [76], explicit dependency management [225], explicit property specification and verification of models [95, 196], explicitly modelling concurrency [174], and the explicitly modelled notion of a concept for operation reuse [237].

1.1.4 Foundation for MPM

The original goal for creating the foundation for MPM was to stimulate further research in MPM. We therefore present several non-trivial contributions to the domain of MPM, explicitly relying on the proposed MPM foundation, thereby highlighting its applicability. To highlight its wide applicability, applications will be given in different research directions, each being a contribution in its own research domain.

These applications both utilise MPM and contribute to MPM. They utilise MPM, as they require concepts such as domain-specific modelling, model transformations, processes, model sharing, and so on, to be effective. They contribute to MPM, as they further aid in the construction of systems following the MPM approach, making it easier to apply and use.

Summary

With the usefulness of MPM motivated, the need for tool support becomes clear, certainly because current tools are lacking. When constructing such a complex tool, the use of MPM techniques is a logical choice, as MPM was assumed to significantly help in the development of complex software. Finally, its usefulness for the application of MPM and future research in MPM has to be evaluated.

1.2 Challenges and Contributions

To fulfill the goals set forth in this thesis, several challenges pose themselves.

- There is no clear set of requirements on what defines an MPM tool. While there is a definition for MPM, it remains relatively abstract and doesn't even go in detail on what a paradigm is exactly. For example, modelling the process was implied in the original definition of MPM, the process being a "relevant aspect", but has been made more explicit in later definitions, as this was rarely done. It is therefore difficult to make sure that all requirements are collected and that the resulting tool is actually a full-fledged MPM tool.

- It is hard to argue for the usefulness of MPM in the context of MPM tool development. While several advantages were attributed to MPM, MPM was never before applied to the construction of an MPM tool itself. This raises questions as to whether it is possible (feasibility), but also on what are the benefits from doing this, if any.

- It remains challenging to define an MPM tool and show that it is both usable in an MPM context today and can be used for further research in MPM in the future. Indeed, it is impossible to predict all future research directions, and it is similarly impossible to know about the problems in all research directions.

Each of these challenges gives rise to new contributions. The following contributions are made in this thesis.

- Our first contribution is a **specification for MPM**, where we explicitly list the requirements for any tool that supports MPM. We base these requirements on the definition of MPM, both implicit and explicit, and on the Power Window case study, which is minimal although representative of a typical application of MPM.

- Our second contribution is that of implementing a prototype tool for this specification by **explicitly modelling all aspects**, using the most appropriate level of abstraction and the most appropriate formalism. For each aspect, we note the (dis)advantages, which can be of interest for other tool builders to decide whether or not to start modelling (parts of) their tool as well. In particular, some explicitly modelled parts resulted in contributions of their own, thereby further building the case for MPM:

  - Explicitly modelling the **conformance relation** allowed for multiple conformance relations to hold simultaneously, potentially with different types of semantics or mappings.

  - Explicitly modelling the **physical type model** allowed for direct access to the physical level of the tool, while remaining independent of it.

  - Explicitly modelling the **external services** allowed for more intuitive control over them, while giving more tools for users to reuse.

  - Explicitly modelling the **process model enactment** allowed for reuse of existing formalisms, and combined with the external services, this has the potential to increase analyzability of processes.

  - Explicitly modelling the **performance** allowed for deterministic performance evaluations supporting what-if scenario's, which might even be executed faster than the actual application.

- Our third contribution is in showing that the tool can be used for the application of and further research on MPM. As to the application of MPM, we present how the Power Window case study is fully supported in our approach, linking back to the original requirements. As to the future research in MPM, we applied our tool to develop and prototype new contributions in several research directions:

  - **Live modelling** was provided in a structured way, making it applicable to many types of domain-specific languages with minimal effort.

- **Concrete syntax** was made more general in terms of supported perceptualization formats (e.g., plots, sound) and the usual icon-mapping restriction was removed.
- **Tool debugging** features were enhanced by changing the level of abstraction at which debugging occurs. This contribution was further enhanced with (efficient) omniscient debugging techniques.

Structure

The remainder of this thesis is structured as follows. Chapter 2 presents the necessary background on various techniques, formalisms, and tools used throughout this thesis. Chapter 3 presents the current state of the art in the various dimensions of MPM. We elaborate on each of these domains and end with a comparison of several existing tools. Chapter 4 presents the target audience and requirements of such a foundation for MPM. Starting from these requirements, a solution architecture and interface is proposed. Chapter 5 presents the various aspects in the development, which use MPM techniques itself. For each component, we discuss the appropriateness of the chosen formalism, after which we present and evaluate the model. Chapter 6 presents several contributions built on top of our foundation for MPM, indicating its extensibility and applicability as a foundation for MPM. These applications include a general framework for live modelling and concrete syntax, and considers a way of debugging the tool. Chapter 7 concludes this thesis and presents future work.

Chapter 2

Background

This thesis makes use of several techniques and formalisms that might require some introduction. We mostly remain at the conceptual level, only introducing the basics of domain-specific modelling and multi-paradigm modelling. In the next chapter, we present state of the art research in these domains and its various branches.

2.1 Domain-Specific Modelling

Developing complex, reactive, software-intensive systems using a traditional, code-centric approach, is not an easy feat: knowledge is required from both the problem domain (*e.g.*, power plant engineering), and computer programming. Apart from being sub-optimal in terms of efficiency and cost, this can result in more fundamental problems. The programmer, implementing the software, has limited knowledge of the problem domain. The domain expert, on the other hand, has deep knowledge of the problem domain, but limited understanding of computer programs. This can result in communication problems, such as false assumptions from either side. Furthermore, the domain experts finally receive a program they don't understand the workings of, making it difficult to validate (and modify). Essentially, the conceptual gap between both domains hinders productivity.

Model-Driven Engineering (MDE) tries to bridge this gap by shifting the level of specification from computing concepts (the "how") to conceptual models or abstractions in the problem domain (the "what"). Domain-Specific Modelling (DSM) [154] in particular makes it possible to specify these models in a Domain-Specific Modelling Language (DSML), using concepts and notations of a specific domain. The goal is to enable domain experts to develop, understand, and verify models more easily, without having to use concepts outside of their domain. It allows the use of a custom visual syntax, which is closer to the problem domain, and therefore more intuitive. Models created in such DSMLs are used, among others, for simulation, (formal) analysis, documentation, and code synthesis for different platforms.

2.1.1 Example

We explain the necessary steps for developing a system using the DSM approach, applied to a simplified nuclear power plant control interface case study. The system consists of two parts: the physical plant and the controller.

Plant The nuclear power plant takes actions as input (*e.g.*, "lower the control rods") and outputs events in case of warning and error situations. Each nuclear power plant component is built according to its specification, which lists a series of requirements (*e.g.*, "the reactor can only hold 450 bar pressure for 1 minute"). These are encoded in the model of each component. Components send warning messages in case their limits are almost reached, such that the user can take control and alleviate the problem. When a component goes outside of its supported boundaries, an emergency shutdown is issued to prevent a nuclear meltdown. There are two types of components:

1. Monitoring components, which monitor the values of their sensors and send messages to the controller depending on the current state of the component. An example is the generator, which measures the amount of electricity generated. Their state indicates the status of the sensors: *normal*, *warning*, or *error*.

2. Executing components, which receive messages from the controller and execute the desired operation. A valve, for example, is either open or closed. Their state indicates the physical state of the component, for example *open* or *closed*.

Controller The controller acts as an interface between the plant and the user: users send messages to the controller by pressing buttons, which the controller translates to operations on the plant. The controller might opt to ignore it or send a different message. For example, when a nuclear meltdown is imminent, the controller might ignore a button press to raise the control rods. We implement a controller which has three main modes:

1. Normal operation, where the user is unrestricted and all messages are passed.

2. Restricted operation, where the user can only perform actions which lower the pressure in the reactor. This mode is entered when any of the components sends out a warning message. When all components are back to normal, full control is returned to the user.

3. Emergency operation, where control is completely taken away from the user. This mode is entered when any of the components sends out an error message. The controller will forcefully shut down the reactor and ignore further input from the user.

We construct a DSML which makes it possible to model such an interface, and express the behaviour of each component, as well as the controller. We intend to automatically synthesize code implementing the modelled behaviour.

2.1.2 Terminology

The first step in the DSM approach when modelling in a new domain is, after a domain analysis, creating an appropriate DSML. A DSML is fully defined [161] by:

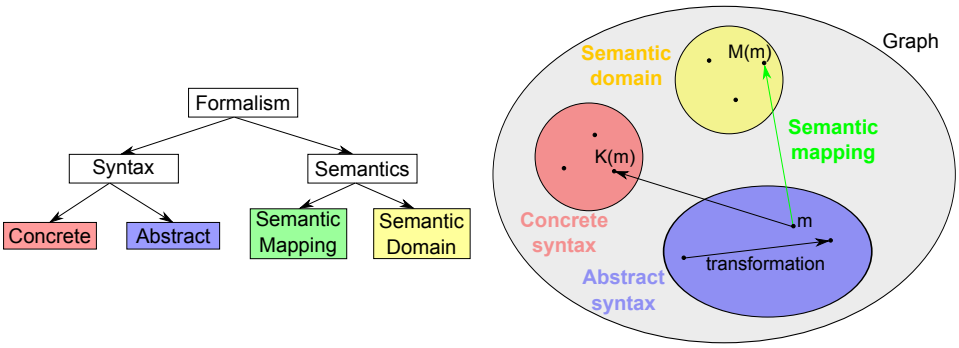


Figure 2.1: Terminology

1. Its **abstract syntax**, defining the DSML constructs and their allowed combinations, typically using a metamodel.
2. Its **concrete syntax**, specifying the visual representation of the different constructs, either graphical (e.g., icons) or textual.
3. Its **semantics**, defining the meaning of models created in the domain [136], encompassing both the **semantic domain** (*what* is its meaning) and the **semantic mapping** (*how* to give it meaning).

This definition of terminology can be seen in Figure 2.1. Each aspect of a formalism is modelled explicitly, as well as relations between different formalisms. We will present these four aspects in detail, presenting the model(s) related to the case study for each.

2.1.3 Syntax

The syntax defines which elements are valid in a specified language, though does not concern itself with what the constructs mean. It consists of two parts: the abstract syntax (**what** constructs are allowed) and concrete syntax (**how** do the constructs look like). The syntax is used later on to generate a domain-specific syntax-directed modelling environment, meaning that, by construction, only valid instances can be created. This maximally constrains the modeller and ensures the models are (syntactically) correct by construction.

Abstract Syntax

The abstract syntax of a language specifies its constructs and their allowed combinations, and can be compared to grammars specifying parsing rules. Such definitions are captured in a metamodel, which are themselves a model of the metametamodel [169]. Most commonly, the metametamodel is similar to UML Class Diagrams, making it possible to define classes, associations between classes (with incoming and outgoing multiplicities), and attributes. While abstract syntax reasons about allowable constructs, it does not state anything about how they are presented to the user.

The abstract syntax definition for the nuclear power plant case study is shown in Figure 2.2. It defines the concepts (e.g., *reactor* and *pump*), relations between them (e.g., a generator cannot be directly connected to a steam valve), and their attributes (e.g., “flow_rate” for a

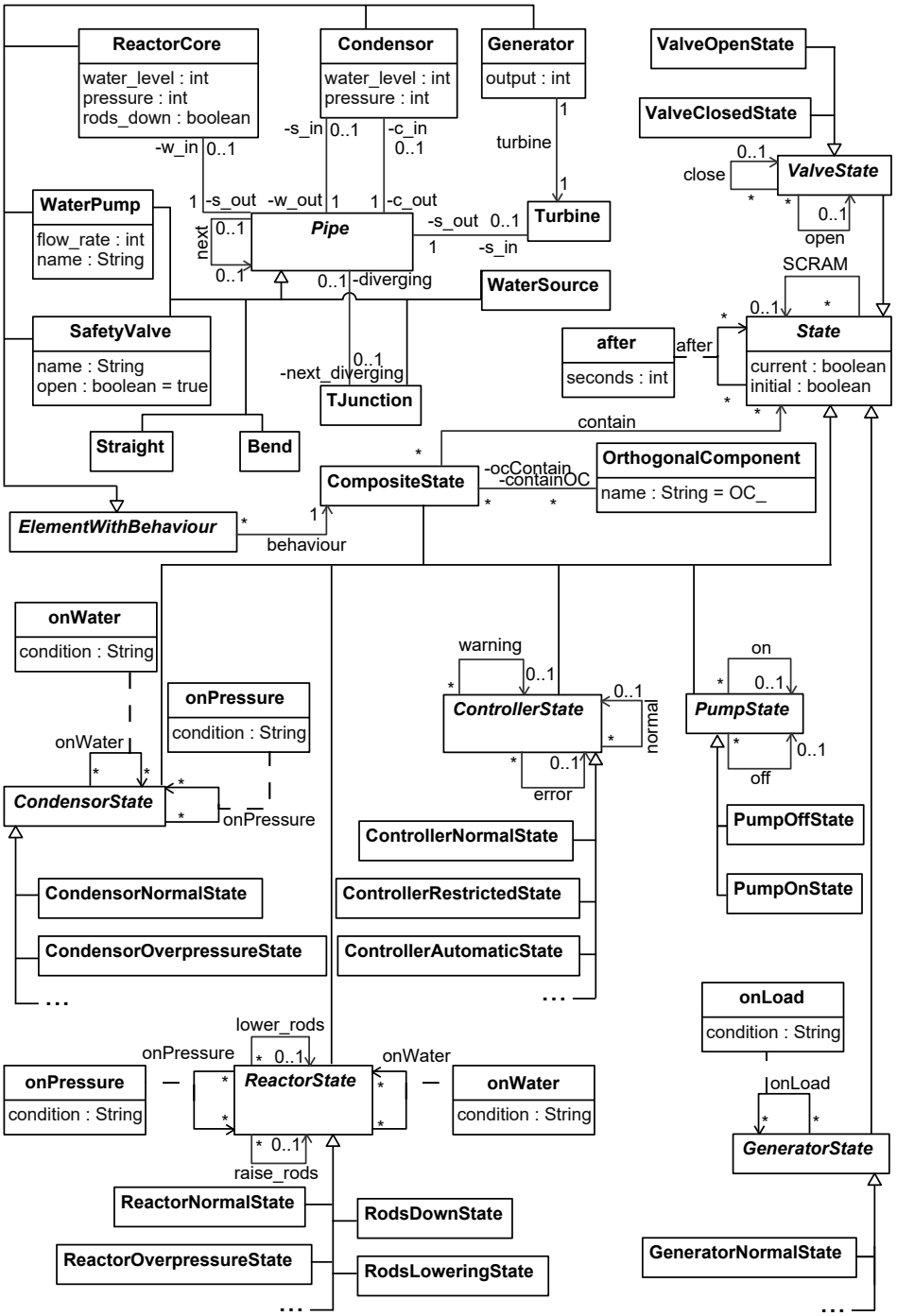


Figure 2.2: Abstract syntax definition for the nuclear power plant domain (some subclasses omitted)

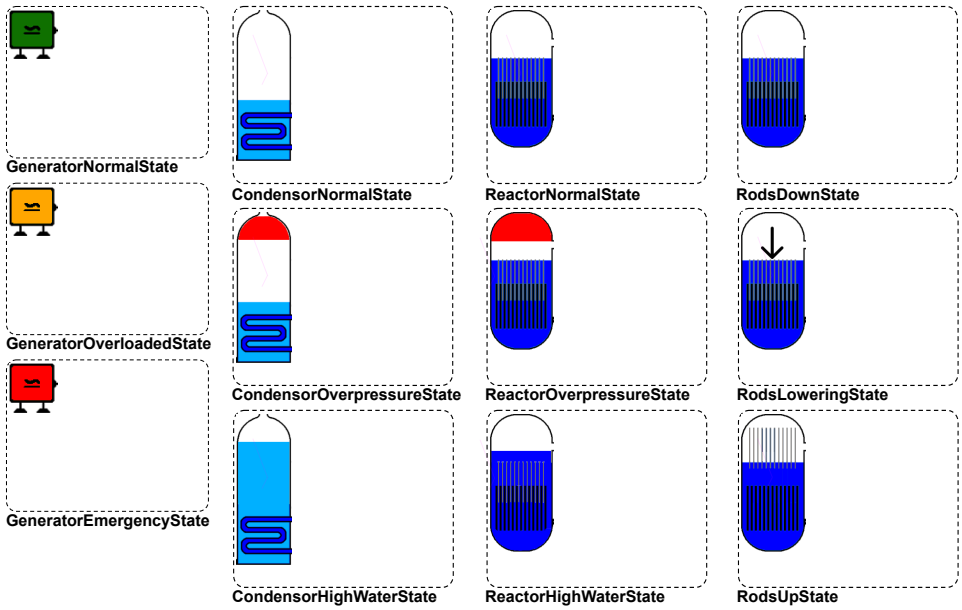


Figure 2.3: Concrete syntax definition for the nuclear power plant domain (excerpt)

pump). For each component, we link it to a behavioural specification, though we haven't defined its semantics (or meaning) yet.

Concrete Syntax

The concrete syntax of a model specifies how elements from the abstract syntax are visually represented. For each representable abstract syntax concept, a concrete syntax mapping is defined. The definition of the concrete syntax is a determining factor in the usability of the DSML, and therefore several "rules" were identified on how to make visual languages usable [198].

Multiple types of concrete syntaxes exist, though they are usually either **textual** or **graphical**. A single model can have different concrete syntax representations, so it is possible for one to be textual, and another to be graphical. Both have their advantages and disadvantages: textual languages are more similar to programming languages, making it easier for programmers to start using the DSML. On the other hand, graphical languages can represent some problem domains better, due to the use of standardized symbols, despite them being generally less efficient [219]. An overview of different types of graphical languages is given in [72].

An excerpt of a possible visual concrete syntax definition for the nuclear power plant case study is shown in Figure 2.3. Each of the constructs presented in the concrete syntax model corresponds to the abstract syntax element with the same name. Now that we have a fully defined syntax for our model, we create an instance of the case study, of which a screenshot is shown in Figure 2.4. Each component additionally has a specification of its dynamic behaviour, specifying when to send out messages, using the state of the underlying system, as well as timeouts.

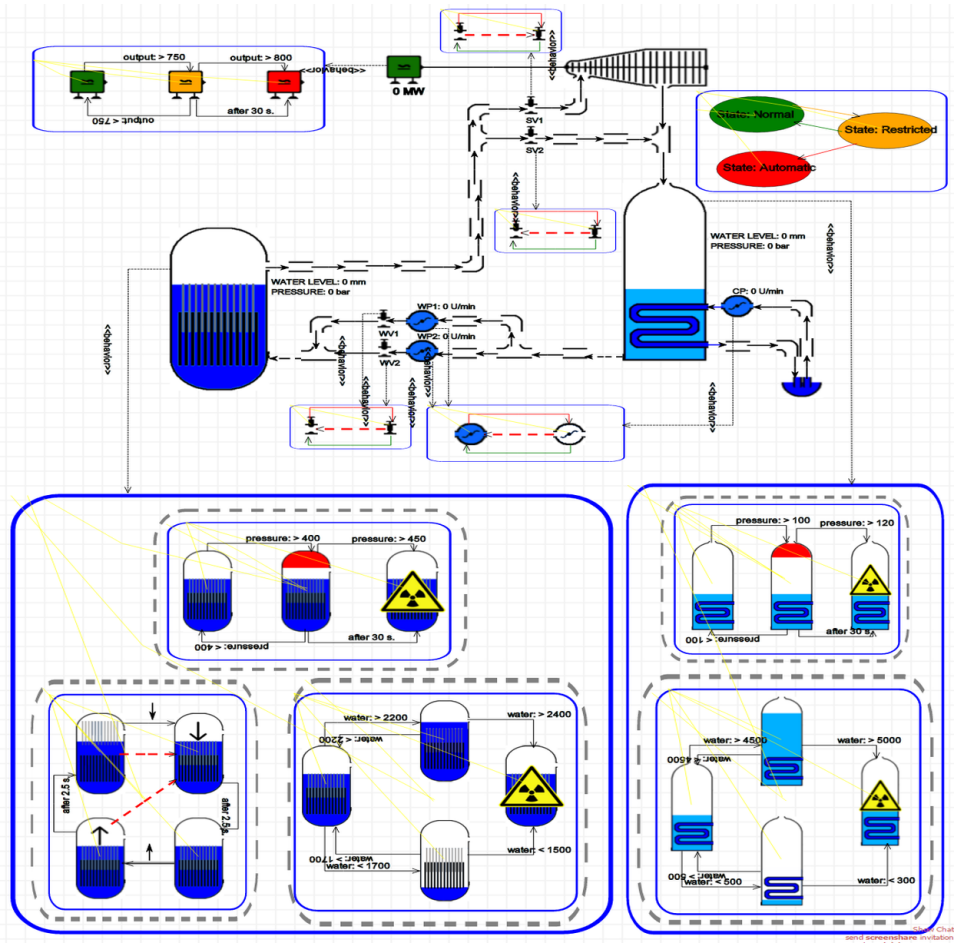


Figure 2.4: Screenshot of an example nuclear power plant instance

2.1.4 Semantics

Since the syntax only defines what a valid model looks like, we need to give a meaning to the models. This is done through semantics, to which there are two parts: the domain it maps to (semantic domain) and the mapping itself (semantic mapping).

Semantic Domain

The semantic domain is the target of the semantic mapping. As such, the semantic mapping will map every valid language instance to a (not necessarily unique) instance of the semantic domain. Many semantic domains exist, as basically every language with semantics can act as a semantic domain. The choice of semantic domain depends on which properties need to be preserved. For example, DEVS [338] can be used for simulation, Petri Nets [203] for verification, Statecharts [133] for code synthesis, and Causal Block Diagrams [61] for continuous systems using differential equations. All these formalisms are elaborated

on later. A single model might even have different semantic domains, each targeted at a specific goal.

For our case study, we use Statecharts as the semantic domain, as we are interested in the timed, reactive, autonomous behaviour of the system, as well as code synthesis. Statecharts were introduced by David Harel [133] as an extension of state machines and state diagrams with hierarchy, orthogonality, and broadcast communication. It is used for the specification and design of complex discrete-event systems, and is popular for the modelling of reactive systems, such as graphical user interfaces. The Statecharts language has defined semantics [135], and can therefore serve as a semantic domain.

Figure 2.5 presents a Statecharts model which is equivalent to the model in our DSML, at least with respect to the properties we are interested in. Parts of the structure can be recognized, though information was added to make the model compliant to the Statecharts formalism. Additions include the sending and receiving of events, and the expansion of forwarding states such as in the controller. The semantic mapping also merged the different behavioural parts into a single Statechart. As is usually the case, the DSML instance is less verbose and more intuitive, compared to the resulting Statechart instance.

Semantic Mapping

While many categories of semantic mapping exist [343], we only focus on the two main categories relevant to our case study: translational and operational semantics.

First, **translational semantics** translates the model from one formalism to another, while maintaining an equivalent model with respect to the properties under study. The target formalism has semantics, meaning that the semantics is "transferred" to the original model. For our case study, this means mapping the model to a Statechart, as presented before.

Second, **operational semantics** effectively executes, or simulates, the model being mapped. Operational semantics can be implemented with an external simulator, or through model transformations that simulate the model by modifying its state. The advantage of in-place model transformations is that semantics are also defined completely in the problem domain, making it suitable for use by domain experts. For our case study, this means implementing a simulator using model transformations.

Both types of semantic mapping are commonly expressed using model transformations, which form the heart and soul of Model-Driven Engineering [257]. A model transformation is defined using a set of transformation rules, and a schedule.

A rule consists of a Left-Hand Side (LHS) pattern (transformation pre-condition), Right-Hand Side (RHS) pattern (transformation post-condition), and possible Negative Application Condition (NAC) patterns (patterns which, if found, stop rule application). The rule is applicable on a graph (the host graph), if each element in the LHS can be matched in the model, without being able to match any of the NAC patterns. When applying a rule, the elements matched by the LHS are replaced by elements of the RHS in the host graph. Elements of the LHS that don't appear in the RHS are removed, and elements of the RHS that don't appear in the LHS are created. Elements can be labelled in order to correctly link elements from the LHS and RHS.

A schedule determines the order in which transformation rules are applied. For our purpose, we use MoTiF [271], which defines a set of basic building blocks for transformation rule

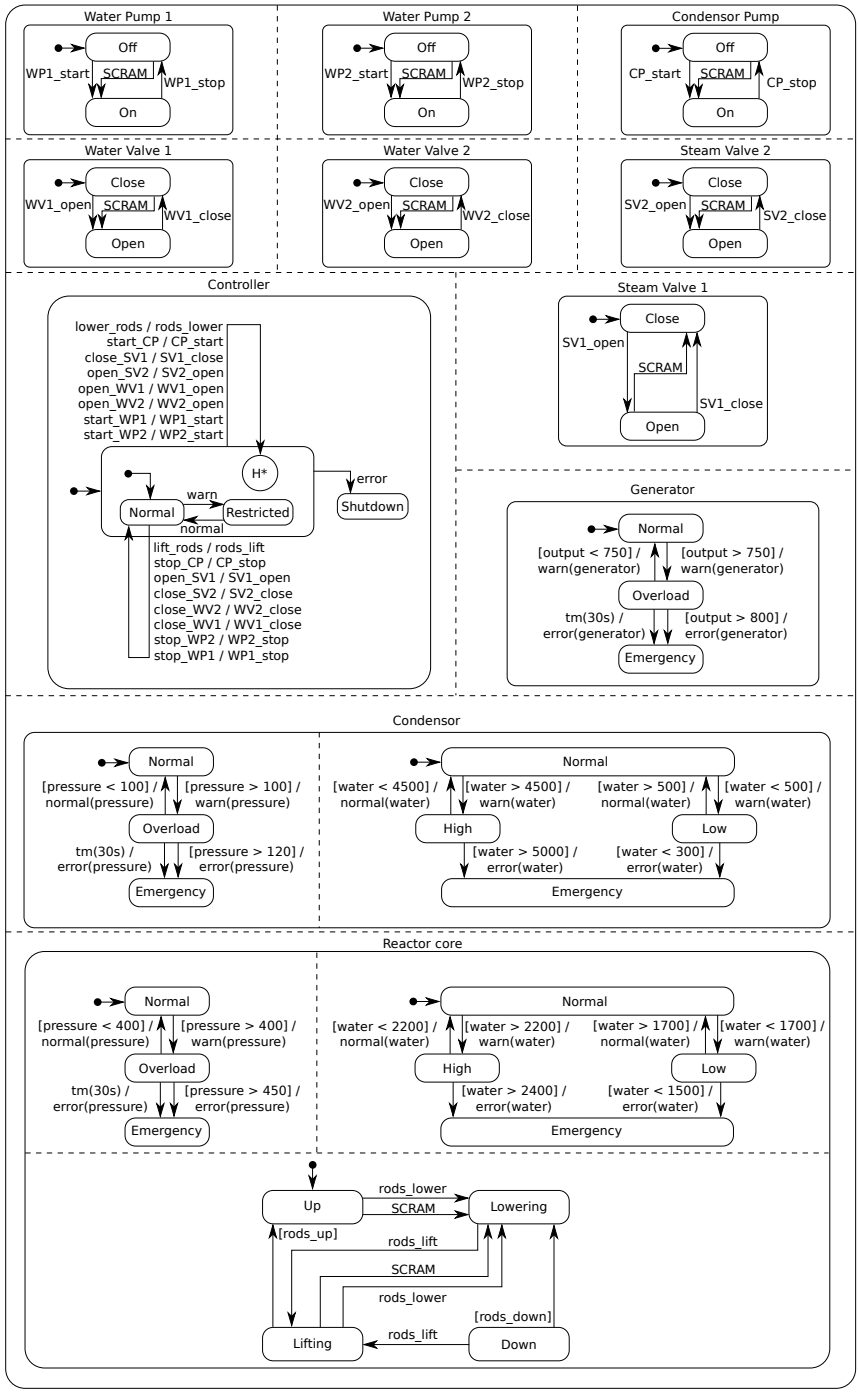


Figure 2.5: Statechart equivalent to the behaviour shown in Figure 2.4

scheduling. We limit ourself to three types of rules: the *ARule* (apply a rule once), the *FRule* (apply a rule for all matches simultaneously), and the *CRule* (for nesting transformation schedules).

Next, we define both operational and translational semantics.

Translational Semantics With translational semantics, the source model is translated to a target model, expressed in a different formalism, which has its own semantic definition. The (partial) semantics of the source model then corresponds to the semantics of the target model. As the rule uses both concepts of the problem domain and the target domain (Statecharts in our case), the modeller should be familiar with both domains.

The schedule of our transformation is shown in Figure 2.6, where we see that each component is translated in turn. Each of these blocks are composite rules, meaning that they invoke other schedules. One such composite schedule is shown in Figure 2.7, where the valves are translated. The blocks in the schedule are connected using arrows to denote the flow of control: green arrows (originating at the checkmark) are followed when the rule executed successfully, while red arrows (originating at the cross) are followed when an error occurred. Three pseudo-states denote the start, the succesful termination, and the unsuccesful termination of the transformation. Our schedule consists of a series of FRules, which translate all different valve states to the corresponding Statecharts states. After these are mapped, the transitions between them are also translated, as shown in the example rule in Figure 2.8. In this rule, we look up the Statecharts states generated from the domain-specific states, based on traceability links. When found, we create a link between them if there was one in the domain-specific language. In this case, we map the SCRAM[1] message to a Statecharts transition which takes a specific kind of event. Note that we do not remove the original model, ensuring traceability.

Operational Semantics With operational semantics, a simulator for that formalism is defined. This simulator can be modelled as a model transformation that executes the model by continuously updating its state (defining a "stepping" function). Contrary to translational semantics, the source model of operational semantics is often augmented with runtime information, creating the difference between a "design formalims" and "runtime formalism". In our case study, for example, the runtime formalism is equivalent to the design formalism augmented with information on current state and simulated time, as well as a list of inputs from the environment.

An example rule is shown in Figure 2.9. The rule changes the current state by following the "onPressure" transition. The left hand side of the rule matches the current state, the value of the sensor, and the destination of the transition. It is only applicable if the condition on the transition (*e.g.*, > 450) is satisfied, by comparing it to the value of the sensor reading. We use abstract states for both source and target of the transition, as we do not want to limit the application of the rule to a specific combination of states: the rule should be applicable for all pairs of reactor states that have an "onPressure" transition. The right hand side then changes the current state to the target of the transition.

[1] A SCRAM is the emergency situation in a nuclear power plant, where the control rods are lowered to effectively stop the nuclear chain reaction.

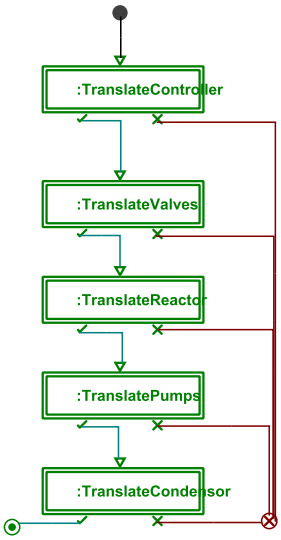


Figure 2.6: Top-level transformation schedule

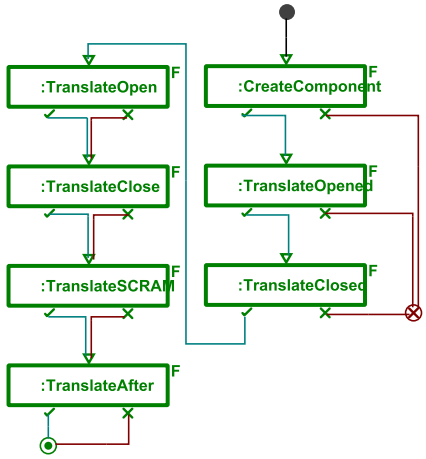


Figure 2.7: Transformation schedule for valves

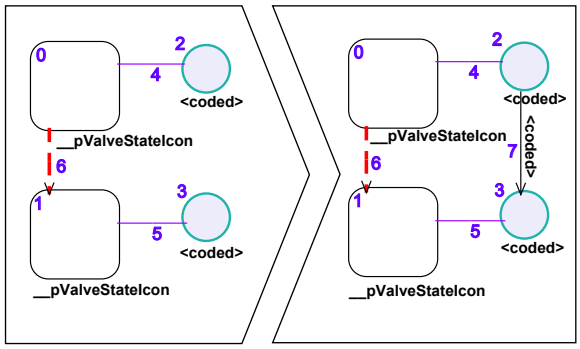


Figure 2.8: Example transformation rule for translational semantics

The schedule has the form of a "simulation loop", but is otherwise similar to the one for translational semantics and is therefore not shown here.

2.2 Process Modelling

The previous discussion of DSMLs already introduced several languages (e.g., Statecharts and the DSL for the nuclear reactor) and a mappings from one language to the other. This quickly becomes convoluted, as indeed it does not stop there: the Statecharts model has semantics, which might be defined by mapping to another formalism, generating code, or something similar. On the other hand, additional operations can be defined on the DSML, such as optimization or expansion of syntactic sugar. For a more complex example, it is only natural that the number of formalisms and activities between them increases as well. This gives rise to the need for a structured approach to follow the required order of operations,

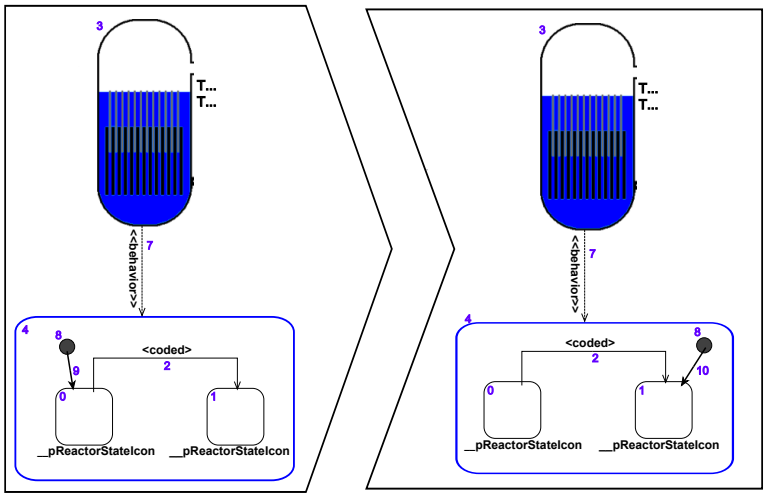


Figure 2.9: Example transformation rule for operational semantics

on the correct models: a process model. This is elaborated on in Section 2.4.7.

2.3 Multi-Paradigm Modelling

Modern engineered, often cyber-physical, systems have reached a complexity that requires systematic design methodologies and model-based aproaches to ensure correct and competitive realization [200]. It is in this context that Multi-Paradigm Modelling (MPM) comes into play: multiple heterogeneous domains come together and need to be managed. MPM proposes to tackle these problems by modelling all relevant aspects of the system explicitly at the most appropriate level(s) of abstraction, using the most appropriate formalism(s), while explicitly modelling the process.

The most appropriate formalism is most often domain-specific, meaning that they can be easily used by domain experts. These various models from different domains are (automatically) mapped to a common domain, which is appropriate for the problem at hand (*e.g.*, Petri Nets). In combination with the process model, the task of domain experts is limited to modelling in the DSMLs they are an expert in, with all manipulations and transformations happening automatically. This shields domain experts from the underlying solution domain, and makes the complete process repeatable, as it can be enacted.

To precisely define MPM, the notion of "paradigm" must be precisely defined. A paradigm P is a collection of formalisms/languages, abstractions, and processes that satisfy properties that are considered characteristic for P. The Object-Oriented paradigm (OOp), for example, includes formalism(s) that have notions of object identity, encapsulation, specialization, etc. Abstractions/formalisms such as UML class diagrams, but also programming languages such as Java satisfy the OOp properties. Processes such as the Rational Unified Process also satisfy OOp properties such as being iterative. With this definition, "multi-paradigm" follows naturally, as a collection of formalisms/languages, abstractions, and processes, composing these of the individual paradigms' properties. This breaks down to composing formalisms/languages, abstractions, and processes.

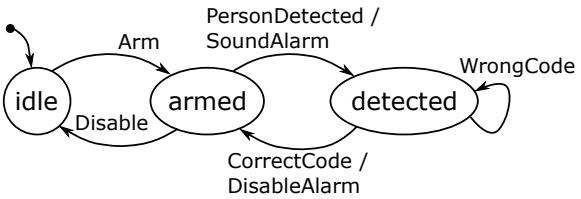


Figure 2.10: Example FSA of a home security alarm system.

MPM combines three complementary research areas:

1. **Language Engineering** to create and instantiate new languages. These languages can be tailored to the problem domain, resulting in Domain-Specific Modelling Languages (DSMLs). This aspect of MPM lowers the cognitive gap between the problem and solution domain by decreasing verbosity and maximally constraining the modeller to the problem at hand.

2. **Activities** to define operations on models, such as to execute them. These activities can be tailored to specific DSMLs, thereby giving semantics to user-defined languages. This aspect of MPM takes models beyond mere documentation, thus increasing their usefulness.

3. **Process Modelling** to define the control and data flow of the development process. The process is tailored to a specific problem, which gives rise to causal dependencies between the used formalisms and activities to map between them. This aspect of MPM provides an overview of the control and data flow used within the process, which can be used for documentation, analysis, optimization, and enactment (i.e., automatic chaining and execution of activities).

2.4 Formalisms

The aforementioned techniques make extensive use of various formalisms, some of which have already been briefly mentioned. The most relevant formalisms, which might require an introduction, are mentioned below.

2.4.1 Finite State Automata

The Finite State Automata (FSA) language [143] is a modelling language used to model reactive systems with discrete state. Its building blocks are:

- *States*, which represent the state a system is in. There is exactly one initial state, where execution starts.

- *Transitions* between states that model the flow of the system. A transition is triggered by an event from the environment, consuming it as the transition is taken. Only transitions whose source state is the current state can fire. After triggering, the target of the transition becomes the new current state. A transition can additionally raise an output event to the environment.

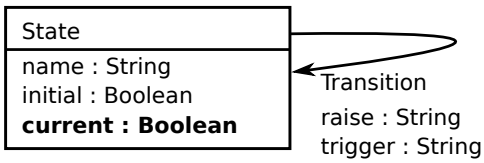


Figure 2.11: Abstract Syntax of Finite State Automata. Runtime-only concepts are shown in bold. The association "Transition" has additional attributes defined on it.

Its abstract syntax is shown in Figure 2.11. A pseudo-code version of its semantics is shown in Algorithm 1.

ALGORITHM 1: FSA operational semantics.

$state \leftarrow s_0$
while $state \notin$ FINALSTATES(M) **do**
  wait for input
  $state \leftarrow$ TARGET$(M, input)$
end while

A frequently used visualization is as a state diagram, where states are represented as circles and transitions as arrows, labelled with their trigger and output event. The initial state is pointed to by an arrow starting from a small black dot. An example is shown in Figure 2.10, where a simple home security alarm system is modelled. In the *idle* mode, the alarm system can be armed by the user. If someone is detected in the *armed* mode, the alarm goes off, until the user inputs the correct combination. The alarm can be disabled by sending the *Disable* event, but only when no intrusion is detected.

2.4.2 Causal Block Diagrams

The Discrete Time Causal Block Diagrams (DTCBD) language [61] is a dataflow language, where signals are propagated through a network of connected blocks. It allows to model systems by defining them as a set of equations. The semantics is given by a set of continuous signals. Blocks implement atomic mathematical operations, which take their input signals and generate, instantaneously, a single output value. The mathematical concepts modelled by these blocks include constants, addition, negation, multiplication, and inversion. Additionally, a delay block is provided which holds the value for a single iteration, thus introducing the notion of "next step". At initialization, the block uses the value coming into it via the Initial Condition (IC) link. Connections between blocks indicate dependencies: the output of the source block is used as input by the target block.

We consider two types of CBDs: discrete time and continuous time.

Discrete Time

The abstract syntax of Discrete Time Causal Block Diagrams (DTCBDs) is shown in Figure 2.12.

Figure 2.13a presents a simple DTCBD model representing the equations shown in Figure 2.13b. The equation for y is reduced to $y = x - y$, which is a direct feedback loop

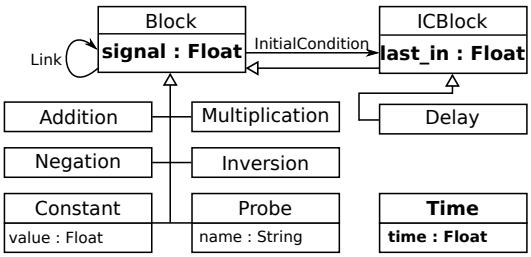
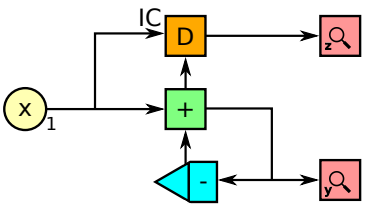


Figure 2.12: Abstract Syntax of Discrete Time Causal Block Diagrams. Runtime-only concepts are shown in bold.

ALGORITHM 2: DTCBD operational semantics.

$clock \leftarrow 0$
$state \leftarrow \text{INITSIGNALS}(M)$
$numIters \leftarrow 0$
while $numIters < maxIters$ **do**
  $g \leftarrow \text{DEPGRAPH}(M, numIters)$
  $s \leftarrow \text{LOOPDETECT}(g)$
  for c **in** s **do**
    if $c = \{gblock\}$ **then**
      $state \leftarrow \text{COMPB}(c, state)$
    else
      $state \leftarrow \text{COMPL}(c, state)$
    end if
  end for
  $clock \leftarrow clock + \Delta t$
  $numIters \leftarrow numIters + 1$
end while
return $clock, state$



(a) Example DTCBD, containing an algebraic loop.

$$\begin{cases} y(t) = x(t) - y(t) \\ z(t) = \begin{cases} x(t) & \text{if } t = 0 \\ y(t-1) & \text{if } t > 0 \end{cases} \end{cases}$$

(b) The equations represented by the example DTCBD model.

Figure 2.13: Example DTCBD.

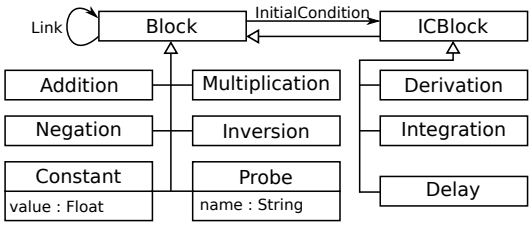
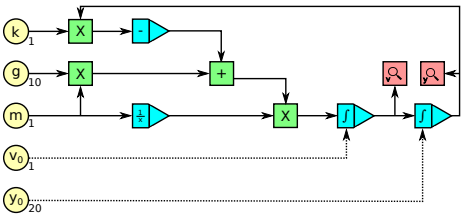


Figure 2.14: Abstract Syntax of Continuous Time Causal Block Diagrams.



(a) Example CTCBD model.

$$\begin{cases} v(t) = \int_0^t \frac{m \cdot g - k \cdot y(t)}{m} dt \\ y(t) = \int_0^t v(t) dt \end{cases}$$

(b) The equations represented by the example CTCBD model.

Figure 2.15: Example CTCBD.

(termed "algebraic loop"). In code, the statement `y = x - y` does not translates to the equation $y = x - y$, as the old values of x and y are used in the right hand side, instead of respecting the feedback loop. To actually implement the equation $y = x - y$, this must be solved to $y = \frac{x}{2}$, which can be implemented in code. Programmers therefore have to (manually) solve the set of linear equations to come up with the code to solve this system of equations. Small changes in the system of equations can result in large changes on the resulting solution. In contrast, DTCBDs handle linear algebraic loops natively, solving $y = x - y$ automatically.

Continuous Time

The Continuous Time Causal Block Diagrams (CTCBD) language [61] is an extension to DTCBDs, introducing two continuous blocks: an integrator and derivator. Its abstract syntax is shown in Figure 2.14. Semantics can be defined by mapping CTCBD models to DTCBD models, thereby discretizing their behaviour. While semantics is not perfectly preserved, this is often a good enough approximation.

Figure 2.15a presents a simple CTCBD model representing the equations shown in Figure 2.15b. These equations model the behaviour of a mass attached to a spring, which is going up and down. When mapping this CTCBD to a DTCBD, the integrator blocks are expanded to a discretized version of the integrator, for example using the forward Euler approach. While both languages look similar, there is a non-trivial translation step between them: discretization. Additionally, while discretizing, it is possible to perform an optimization step for constant folding, dead-block removal, and flattening [211]. Two example model transformation rules are shown for expansion (Figure 2.16) and optimization (Figure 2.17).

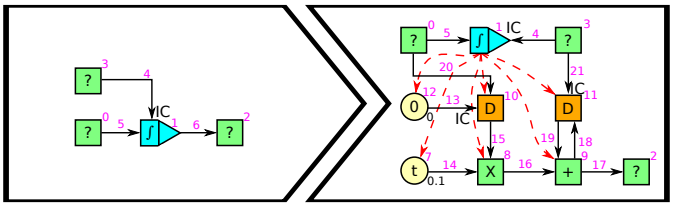


Figure 2.16: Expand rule for an integrator block.

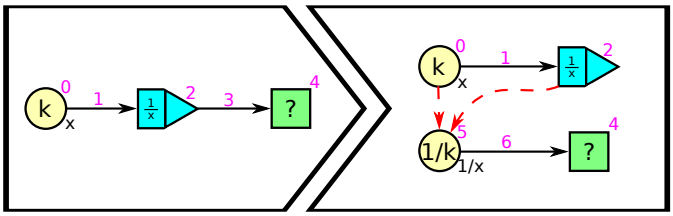


Figure 2.17: Optimize rule for an invertor block.

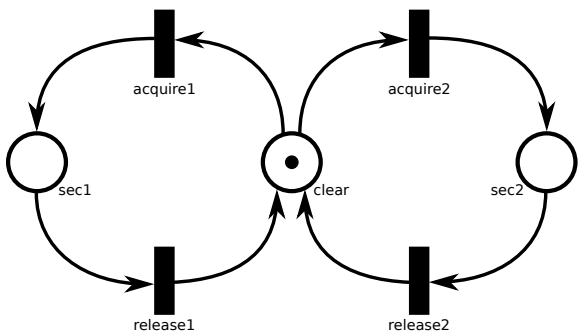
2.4.3 Petri Nets

The Petri Nets (PN) formalism is a formalism used to model non-determinism and concurrency. A Petri Net is a directed, weighted bipartite graph, consisting of two types of nodes: places and transitions. Places are marked with a number of tokens, which together form the initial marking of the Petri Net. The number of black dots (or a number) in a place denotes the marking of that place. Edges can be drawn from a place to a transition, or vice versa. Arcs are annotated with a weight, though no annotation means weight one. Having n arcs between the same place and transition is similar to a single arc with weight n. Visually, a place is represented as a circle, whereas a transition is represented as a bar.

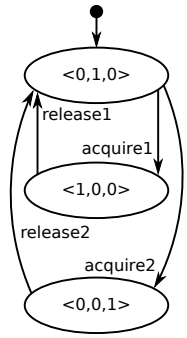
A transition is considered to be *enabled* if, for all incoming arcs, the weight is less than or equal to the marking of the place it originates from. An enabled transition can be *fired*, in which case the markings of all connected places are updated. For the incoming arcs, the connected places have their marking *decremented* with the weight of the arc. For the outgoing arcs, the connected places have their marking *incremented* with the weight of the arc.

An example Petri Net model is shown in Figure 2.18a, modelling a simple critical section protected by a semaphore. The initial marking of the semaphore is one (indicating that it is available), while the critical sections both have zero tokens (indicating that they are not entered). Both "acquire" transitions can be fired, but after one is fired, the other becomes disabled, as the semaphore token is consumed. Afterwards, only the "release" transition can be fired of the same branch that executed its "acquire" transition.

The primary applications of Petri Nets is for analyzability: reachability, boundedness, liveness, deadlocking, and so on. [203] Many of these properties are analyzed using a reachability graph, containing all markings reachable from the initial marking, interconnected with the transitions that need to be fired to go from one marking to another. For example, the reachability graph of the example model in Figure 2.18a is presented in Figure 2.18b. Analyzing this reachability graph, it is clear that it is impossible to have a



(a) Petri Nets example model of a mutual exclusion algorithm for two critical sections.



(b) Reachability graph of the Petri Net shown in Figure 2.18a. Each state encodes the number of tokens in the places as follows: $\langle sec1, clear, sec2 \rangle$

Figure 2.18: Example Petri Net with reachability graph.

token in both critical sections simultaneously (i.e., both critical sections cannot be entered simultaneously).

2.4.4 Statecharts

The Statecharts (SC) formalism is an extension of state machines and state diagrams, with hierarchy, orthogonality, and broadcast communication. It is popular for the modelling of reactive systems, such as graphical user interfaces. Its basic building blocks are *states* that are connected to each other through *transitions*, encoding how the system behaves: a transition specifies to which event it conditionally reacts, and optionally which event should be raised when the transition is fired. *Hierarchical states* group a number of states, of which one is the default. When the hierarchical state is entered, its default state is entered as well. *Orthogonal states* specify behaviour that is executed concurrently and are contained by a hierarchical state. Its semantics are defined by flattening the orthogonal components (*i.e.*, taking the cross products of the states of all hierarchical components). Different orthogonal components communicate with each other through events.

A Statechart generally consists of the following elements:

- states, either basic, orthogonal, or hierarchical;
- transitions between states, either event-based or time-based;
- actions, executed when a state is entered or exited;
- guards on transitions, modelling conditions that need to be satisfied in order for the transition to "fire";
- history states, a memory element that allows the state of the Statechart to be restored.

2.4.5 Statecharts + Class Diagrams

The Statecharts + Calss Diagrams (SCCD) [298] formalism extends Class Diagrams by associating each class with a definition of its behaviour in the form of a Statecharts model. We extend SCXML for the modelling of SCCD models, thus creating the SCCDXML language [298]. Although it is background to our approach, this language was (co-)created within the scope of this thesis. We first present the new features of our language, and then we discuss the management of objects at runtime.

Despite SCCD being an extension of the Class Diagrams language, several additions to the Statecharts formalism were made as well. These additions allow the Statecharts to interact with the Class Diagrams and handle novel situations occuring due to their combination. Additions to both Class Diagrams and Statecharts include:

- ports, which allow the model to communicate with its environment;
- default class, which is the class that is instantiated at startup;
- library imports, allowing the use of programming language libraries within the Statecharts;
- relationships, allowing associations and inheritance between different classes;
- additional event scopes, such as *local* (only visible in the sending Statecharts), *broad* (visible in all Statecharts instances), *output* (visible on the output ports), *narrow* (visible in a single target Statecharts), and *cd* (class diagram management event).

At runtime, a central entity called the object manager is responsible for creating, deleting, and starting class instances, as well as managing links (instances of associations) between class instances. It also checks whether no minimal or maximal cardinalities are violated when the user deletes or instantiates an association, respectively. As mentioned previously, instances can send events to the object manager using the "cd" scope. The object manager can thus be seen as an ever-present, globally accessible object instance, although it is implicitly defined in the runtime, instead of as an SCCD class.

When the application is started, the object manager creates an instance of the default class and starts its associated Statecharts model. From then on, instances can send several events to the object manager to control the set of currently executing objects. The object manager accepts four events: *create_instance*, *delete_instance*, *start_instance*, and *associate_instance*.

The object manager, in combination with input/output ports of the diagram, replaces the *invoke* and *send* tags of the current SCXML standard. We believe this solution to be more general and more modular. The *invoke* tag, for example, does not allow for instances (effectively, agents) to run concurrently for the whole duration of the program, does not offer a comprehensive interface for object management, and does not offer any checks on the structure of the system at runtime. Moreover, using ports instead of direct sends to a predefined location is more modular, since the Statecharts model does not need to know the actual service that it communicates with (it just needs to know its interface), which means it can be reused in different contexts.

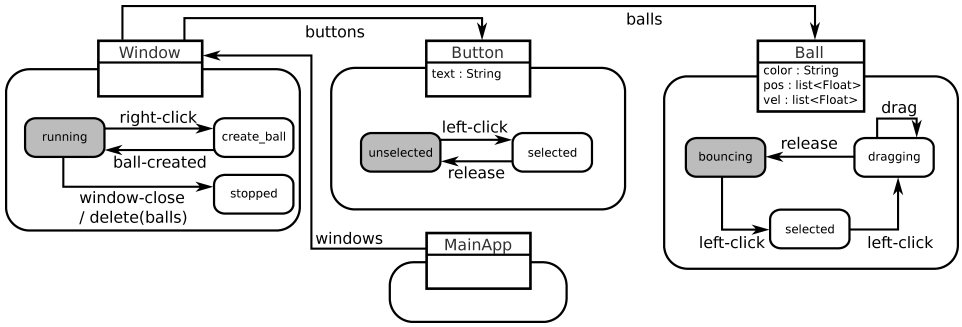


Figure 2.19: Bouncing ball example in SCCD (abstracted).

Figure 2.19 presents a simple bouncing ball example as an SCCD model. It is shown that there are four classes: *MainApp* (the default class), *Window* (representing a window in which the ball is bouncing), *Button* (buttons in the window), and *Ball* (the ball itself). Apart from the *MainApp*, each class has some associated behaviour. For example, a ball can be bouncing, selected, or being dragged, and switches between these modes with mouse clicks.

2.4.6 Discrete Event System Specification

DEVS is a discrete-event formalism with a long history, which started in 1976 [338]. Since then, many variants have come to exist, all specialized for a specific domain. Main advantages of DEVS are its rigorous and precise specification, as well as its support for modular, hierarchical construction of models. DEVS frequently serves as a simulation "assembly language" to which models in other formalisms are translated, either giving meaning to new (domain-specific) languages, or reproducing semantics of existing languages. Models in different formalisms can hence be meaningfully combined by mapping them onto DEVS.

A more elaborate explanation of DEVS is given in our DEVS tutorial [312].

DEVS comprises two types of models: Atomic DEVS models (defining behaviour) and Coupled DEVS models (defining structure).

Atomic DEVS

An Atomic DEVS model is the basic building block of a DEVS model. Its specification is shown in Specification 2.1.

$$AM = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{2.1}$$

X	*set of input events*
Y	*set of output events*
S	*set of sequential states*
$\delta_{int} : S \to S$	*internal transition function*
$\delta_{ext} : Q \times X \to S$	*external transition function*
$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$	*set of total states*
$\lambda : S \to Y \cup \{\phi\}$	*output function, with ϕ the null event*
$ta : S \to \mathbb{R}^{+}_{0,+\infty}$	*time advance*

Intuitively, the behavioural semantics are as follows. The system enters a sequential state $s \in S$ and schedules an "internal" transition to state $\delta_{int}(s)$ after $ta(s)$. Before undergoing the transition, $\lambda(s)$ is invoked to generate an output event $y \in Y$. If before the scheduled internal transition occurs, an external input event $x \in X$ is received, the scheduled output generation and subsequent transition do not occur. Instead, an "external" transition is made to $\delta_{ext}((s, e), x)$. Here, e is the elapsed time, the time that has passed in the state s since the last transition, until the event was received. No output is generated in this case. Upon arrival in the new state, either through δ_{int} or δ_{ext}, the algorithm repeats.

Coupled DEVS

A Coupled DEVS model is the structuring concept of DEVS, and allows various Atomic and Coupled DEVS models to be combined through parallel composition. Its structure is shown in Specification 2.2.

$$CM = \langle X_{self}, Y_{self}, D, MS, IS, ZS, select \rangle \tag{2.2}$$

X_{self}	*set of input events*
Y_{self}	*set of output events*
D	*set of model instance labels*
$MS = \{M_i \mid i \in D\}$	*set of submodels*
$M_i = \{\langle X_i, Y_i, S_i, \delta_{int,i}, \delta_{ext,i}, \lambda_i, ta_i \rangle \mid i \in D\}$	*(atomic) submodel specification*
$IS = \{I_i \mid i \in D \cup \{self\}\}$	*influencee mapping (topology)*
$I_i = 2^{D \cup \{self\} \setminus \{i\}}$	*set of influencees' labels*
$ZS = \{Z_{i,j} \mid i \in D \cup \{self\}, j \in I_i\}$	*translation mapping*
$Z_{self,j} : X_{self} \to X_j$	*input-to-input translation*
$Z_{i,j} : Y_i \to X_j$	*output-to-input translation*
$Z_{i,self} : Y_i \to Y_{self}$	*output-to-output translation*
$select : 2^D \to D$	*select function*

Intuitively, the semantics are given as follows. The coupled DEVS model instantiates all of its submodels, which are all considered to be atomic DEVS models (as a coupled model can always be flattened to an atomic model), and keeps their references (or labels) in D. The atomic DEVS submodels' specifications are found in MS. Submodels can be connected and the connection topology is encoded in the influencee set IS, which lists for each subcomponent, all the other subcomponents that it influences (i.e., sends its output to). Upon forwarding an event to another subcomponent, the event is translated by ZS, which can map input-to-input (for external input coupling), output-to-input (for internal coupling), and output-to-output (for external output coupling). When multiple internal events are scheduled at the same time, in different sub-models, the *select* function is invoked for tie-breaking.

Example

The DEVS specification of a simple traffic light model is given below. Note that, as per the DEVS semantics, the output function λ is invoked *before* the internal transition δ_{int} is taken. This explains why the produced events are non-intuitive (e.g., raise *show_yellow* for GREEN). The traffic light atomic DEVS model is shown in Specification 2.3. The policeman's behaviour is simpler and is shown in Specification 2.4. Finally, the atomic models are composed in a coupled DEVS model, shown in Specification 2.5. From this complete DEVS specification example, it is clear that something is lacking: nowhere has been specified what is the initial state, and how long the system has already been in that state.

Initial Total State

While the previous definition of Atomic and Coupled DEVS models is as in the literature, it does not include initialization [317]. Initialization is often left to the tool, though many implementations disagree on how this is done, as it is not a part of the formalism. For example, while some tools allow the initialization of the initial state and how long the simulator remains in this state, before transitioning, other simulators only take the initial state. Both situations can lead to problems. When only the initial state is given (e.g., adevs [210], VLE [226], and X-S-Y [144]), modellers don't have the required flexibility: they cannot configure how long the model is already in that state. When both the initial state and an initial time advance is given (e.g., CD++ [328], DEVS-Suite [158], MS4Me [258], PowerDEVS [39]), this problem is addressed, although inconsistencies might arise. Indeed, the initial time advances bears no relation to the actual time advance, meaning that it is perfectly possible to stay in the initial state far longer than its time advance has dictated. Given that the external transition function explicitly requires that $e \leq ta(s)$, this could be violated, without any protection from the tool or the formalism. In the light of DEVS standardization efforts [10, 251], it is problematic that various tools implement different methods of initialization.

To address these problems, we augment the atomic DEVS specification with an initial total state $q_{init} \in Q$, comprising the initial state s_{init} and the initial elapsed time e_{init}. This alters the atomic DEVS specification to the following.

$$AM = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

$$Light = \langle X_{light}, Y_{light}, S_{light}, \delta_{int,light}, \delta_{ext,light}, \lambda_{light}, ta_{light} \rangle \tag{2.3}$$

$$\begin{aligned}
X_{light} &= \{\textit{toAuto}, \textit{toManual}\} \\
Y_{light} &= \{\textit{show_green}, \textit{show_yellow}, \textit{show_red}, \textit{turn_off}\} \\
S_{light} &= \{\text{GREEN}, \text{YELLOW}, \text{RED}, \text{GOING_MANUAL}, \text{GOING_AUTO}, \text{MANUAL}\} \\
\delta_{int,light} &= \{\text{GREEN} \rightarrow \text{YELLOW}, \text{YELLOW} \rightarrow \text{RED}, \text{RED} \rightarrow \text{GREEN}, \\
&\quad \text{GOING_MANUAL} \rightarrow \text{MANUAL}, \text{GOING_AUTO} \rightarrow \text{RED}\} \\
\delta_{ext,light} &= \{(\text{GREEN}, _, \textit{toManual}) \rightarrow \text{GOING_MANUAL}, \\
&\quad (\text{YELLOW}, _, \textit{toManual}) \rightarrow \text{GOING_MANUAL}, \\
&\quad (\text{RED}, _, \textit{toManual}) \rightarrow \text{GOING_MANUAL}, \\
&\quad (\text{MANUAL}, _, \textit{toAuto}) \rightarrow \text{GOING_AUTO}\} \\
\lambda_{light} &= \{\text{GREEN} \rightarrow \textit{show_yellow}, \text{YELLOW} \rightarrow \textit{show_red}, \\
&\quad \text{RED} \rightarrow \textit{show_green}, \text{GOING_MANUAL} \rightarrow \textit{turn_off}, \\
&\quad \text{GOING_AUTO} \rightarrow \textit{show_red}\} \\
ta_{light} &= \{\text{GREEN} \rightarrow 57, \text{YELLOW} \rightarrow 3, \text{RED} \rightarrow 60, \\
&\quad \text{MANUAL} \rightarrow +\infty, \text{GOING_MANUAL} \rightarrow 0, \text{GOING_AUTO} \rightarrow 0\}
\end{aligned}$$

$$Police = \langle X_{police}, Y_{police}, S_{police}, \delta_{int,police}, \delta_{ext,police}, \lambda_{police}, ta_{police} \rangle \tag{2.4}$$

$$\begin{aligned}
X_{police} &= \{\} \\
Y_{police} &= \{\textit{toAuto}, \textit{toManual}\} \\
S_{police} &= \{\text{BREAK}, \text{WORKING}\} \\
\delta_{int,police} &= \{\text{BREAK} \rightarrow \text{WORKING}, \text{WORKING} \rightarrow \text{BREAK}\} \\
\delta_{ext,police} &= \{\} \\
\lambda_{police} &= \{\text{BREAK} \rightarrow \textit{go_to_work}, \text{WORKING} \rightarrow \textit{take_break}, \\
ta_{police} &= \{\text{BREAK} \rightarrow 120, \text{WORKING} \rightarrow 360\}
\end{aligned}$$

$$System = \langle X_{self}, Y_{self}, D, MS, IS, ZS, select \rangle \tag{2.5}$$

$$\begin{aligned}
X_{self} &= \{\textit{toAuto}, \textit{toManual}\} \\
Y_{self} &= \{\textit{show_green}, \textit{show_yellow}, \textit{show_red}, \textit{turn_off}\} \\
D &= \{light1, police1\} \\
MS &= \{M_{light1} = Light, M_{police1} = Police\} \\
IS &= \{light1 \rightarrow \{self\}, self \rightarrow \{light1\}, police1 \rightarrow \{light1\}\} \\
ZS &= \{Z_{self,light1} = \{\textit{toAuto} \rightarrow \textit{toAuto}, \textit{toManual} \rightarrow \textit{toManual}\}, \\
&\quad Z_{police1,light1} = \{\textit{take_break} \rightarrow \textit{toAuto}, \textit{go_to_work} \rightarrow \textit{toManual}\}, \\
&\quad Z_{light1,self} = id \\
select &= \{\{light1, police1\} \rightarrow police1, \{ligth1\} \rightarrow light1, \{police1\} \rightarrow police1\}
\end{aligned}$$

Initial State The initial state $s_{init} \in S$ specifies the system state in which simulation commences. Its addition is logical, and has up to now been implemented in different simulation tools, as an implicit ad-hoc extension to the formalism. In the case of the traffic light example, we may specify that the traffic light starts in the GREEN state and the policeman in the BREAK state.

Initial Elapsed Time The initial elapsed time e_{init} specifies how long the system has been in this state, without a transition being observed. We argue for its importance in providing flexibility to the DEVS modeller.

If only an s_{init} is present in the specification, but not e_{init}, it is possible to specify GREEN and BREAK as the initial state for the traffic light and policeman, respectively, but one is restricted to their time advances. Indeed, without e_{init}, the initial elapsed time would be implicitly equal to 0. This schedules the first internal transition of the policeman at time 120. The traffic light will have internal transitions at times 57 (to YELLOW), 60 (to RED), and 120 (to GREEN). Following this sequence, the policeman will *always* send its interrupt at the exact same point in time, namely when a switch is made from RED to GREEN. Therefore, it is impossible for the modeller to reproduce the real-world scenario where, for example, the policeman interrupts after the light has been in the YELLOW state for 1.5 time units. To do this, the model itself would have to be drastically modified (e.g., adding an artificial "initialization" state to the policeman model). This impedes modular re-use of submodels.

What we wish to achieve is shown in Figure 2.20, which includes the "negative" simulation time (i.e., what hypothetically happened before simulation, given the specified model). While this figure includes the state trace from before the start of the simulation, we can only go back until the last transition, as we have no knowledge about how we ended up in that state (e.g., before time -10 for *police1*). To remain consistent with the DEVS specification, it is required to alter the duration since the last event for each atomic model individually. By doing this, each individual atomic DEVS model can be shifted relatively to all others. For example, Figure 2.21 presents two different initial elapsed time configurations, with their effect on the simulation.

In the traffic light example, we augment the models with the following initial total states. For the sake of the closure under coupling, we set the initial elapsed time of the traffic light to 10. Therefore, the light enters YELLOW at time 47. To ensure that the policeman sends the external interrupt after the light has been in state YELLOW for 1.5, the transition happens at time 48.5. To achieve this, we compute its desired initial elapsed time as $e_{init,police1} = 120 - 48.5 = 71.5$. Thus, the policeman has spent 71.5 time units in BREAK, and transitions after 48.5 time units, at which point the light will have been in YELLOW for 1.5 time units, as desired. Various configurations exist to achieve the same result, such as having the traffic light start at a different state or with a different elapsed time.

$$q_{init,light1} = (\text{GREEN}, 10) \qquad q_{init,police1} = (\text{BREAK}, 71.5)$$

2.4.7 Formalism Transformation Graph + Process Model

The Formalism Transformation Graph + Process Model (FTG+PM) [186] formalism was conceived to provide an overview of the complete process, control flow and data flow,

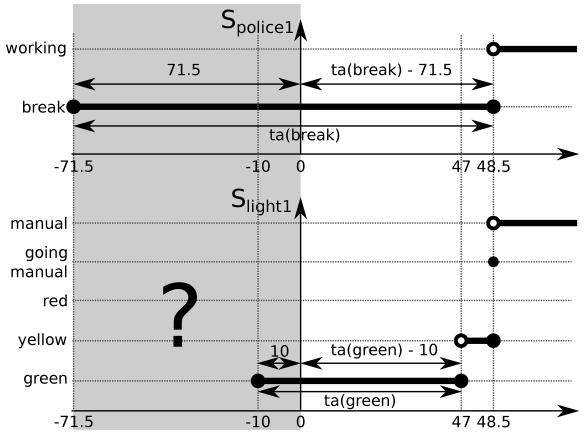


Figure 2.20: Simulation trace with hypothetical negative simulation time (grayed out).

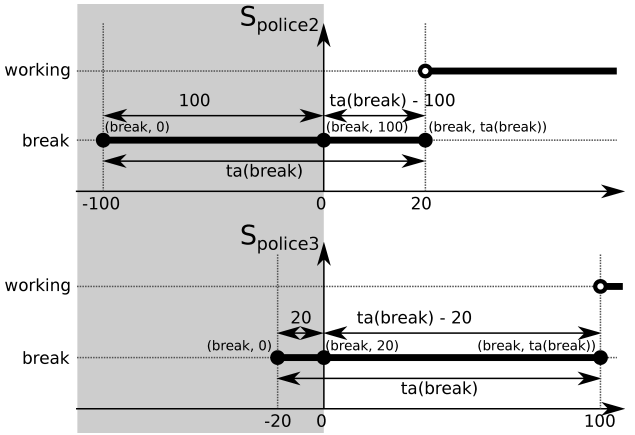


Figure 2.21: Various options for shifting the police model.

including all used formalisms and the activities between them. As the name suggests, an FTG+PM model consists of two parts: the Formalism Transformation Graph (FTG) and the Process Model (PM). The FTG presents all used formalisms (the nodes) and their relations (the links), and presents an overview of all languages and activities used in the process. The PM mandates the order of activities to execute, as well as on which models (data) they operate.

An FTG+PM model has two primary applications: documentation and enactment. As documentation, the FTG+PM summarizes the languages used, the activities between them, and the process of getting from start to finish, including control and data flow. Enacting an FTG+PM means to execute the series of activities defined in the process model in the correct order: start at the initial node and execute the activities in the defined order. There might be concurrent activities, which are then executed concurrently. Execution creates or modifies the models linked to that activity.

For example, consider the safety analysis of a simplified power window [200]. All modelling

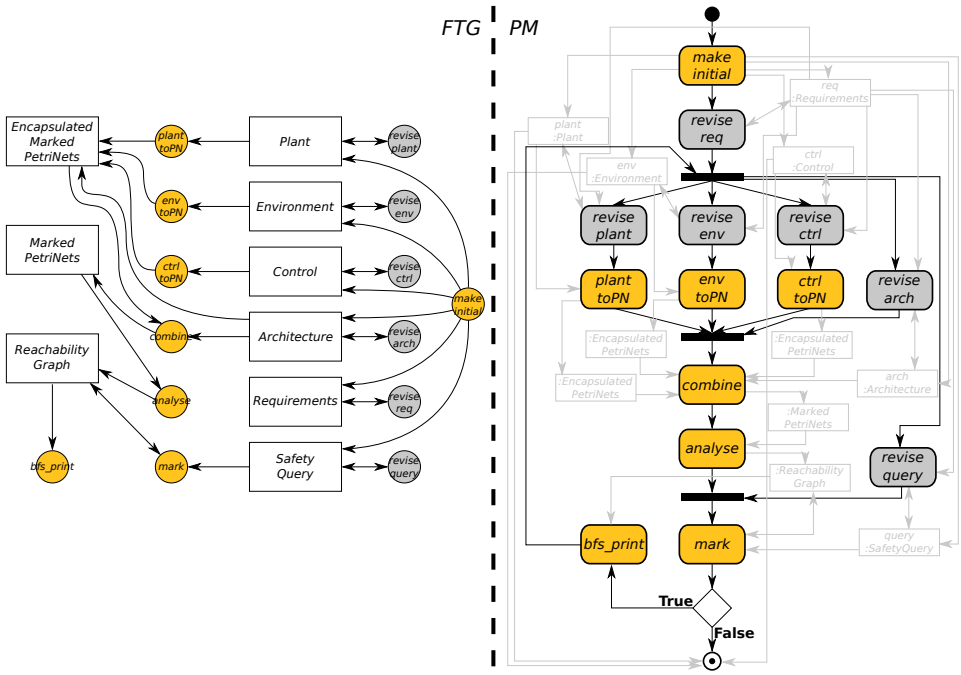


Figure 2.22: Example FTG+PM for safety analysis of a simplified power window.

happens using domain-specific languages, as this is done by domain experts. The actual safety analysis happens using Petri Nets and by creating a reachability graph, which is verified to make sure that an unsafe state can never be reached. Domain-specific languages are thus translated to Petri Nets, transparent to the domain experts.

Figure 2.22 presents the FTG+PM of this simplified process. A more complete FTG+PM is discussed elsewhere [91, 204]. On the left side, all formalisms are shown (rectangles), including the domain-specific ones (e.g., *Plant*, *Environment*, *Control*) and the general purpose ones (e.g., *Marked Petri Nets*, *ReachabilityGraph*). The activities between them are also shown (circles) and can be either automated (e.g., *PlantToPN*, *Combine*) or manual (e.g., *RevisePlant*, *ReviseQuery*). For each activity, it is shown which type of models it takes as input and which it returns as output. For example, *mark* takes in a *ReachabilityGraph* and *SafetyQuery* and outputs a new *ReachabilityGraph*. On the right side, the process is shown, which starts at the top, and uses the formalisms and activities on the left. Execution starts at the top of the PM, which is in this case an Activity Diagram. Besides control flow, there is also a notion of data flow: each activity generates a set of models, which can then be reused. The models on the right side are instances of the formalisms on the left side. When the fork node is reached, all five branches execute concurrently. When the join node is reached, control only progresses when all connected branches have terminated. In this case, we see that there are DSLs for all aspects of the process, after which the translation to *Petri Nets* and the reachability analysis happen automatically. In other words, the domain experts are never shown the Petri Nets on which the analysis was actually done.

Chapter 3

State of the Art

Given the necessary background, we now introduce the reader to several relevant aspects of MPM in full detail. For each aspect, often a research domain on itself, we briefly mention some of the ongoing research in these domains. The full extent of these domains is not necessary to achieve MPM, though it aids in an MPM context as well. At the end of this chapter, several tools are compared according to their support for these aspects of MPM. As will be shown, each tool focusses on a select number of research domains, though they go deep in them. In contrast to these tools, our prototype tool for MPM touches upon all mentioned research domains, but not at the same level of depth.

3.1 Language Engineering

In the domain of language engineering, one particular focus is the abstract syntax and the associated conformance relation. Of particular interest to this section is the definition of the abstract syntax: the *metamodel*. As it is the abstract syntax model that constraints its instances, thus defining the type, it is often termed the *type model*. It is through such metamodels that new languages are engineered. In most cases, a metamodel consists of two parts [256]: 1) a structural constraint, which is itself typed by another type model, and 2) static semantics, which consists of a set of expressions in a constraint language such as OCL [3]. Static semantics, although not related to semantics, are used to further constrain the set of correct instances, as shown in Figure 3.1. A model is expressed as a complex graph of objects [266, 267], which conforms to the specified type model. As both parts are merely constraints, a type model is in essence a set of constraints, on which constraint satisfaction can be used to automatically generate instances [256]. We elaborate on the single most important aspect of every (meta-)modelling tool: the type/instance relation [32, 43, 169, 279].

3.1.1 Instantiation and Conformance

Figure 3.1 naturally raises the question how to go from the type model to the instance, and back. For this, two particular functions are defined: *instantiation* and *conformance*.

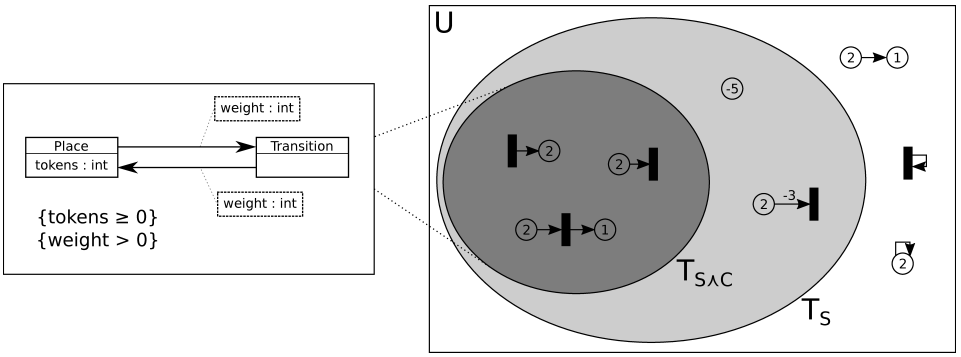


Figure 3.1: Meaning of a simple Petri Nets metamodel. U represents all possible instances, including those violating the structure; T_S represents all instances that conform structurally, but not necessarily to all the constraints; $T_{S \wedge C}$ represents all instances that conform both structurally and fulfil all constraints.

Instantiation starts from a type model, and generates an instance of that particular type model. For example, instantiating a *Place* from the Petri Net metamodel returns a *Place* instance, visualized as a circle. The particular configuration of this element (*i.e.*, its number of tokens) is not defined: it can be set to a hard-coded default value, a user-defined default value, or simply a positive random number. Whichever option is taken, depends on the instantiation semantics defined over this type model.

Conformance goes the other way around, checking whether or not a particular object is an instance of the type it specifies. Generally, a model should conform to its metamodel when performing operations on it, as otherwise the model is considered invalid. Depending on the conformance check, it might be necessary to pass a type mapping too: a way to link elements from the instance to type it is instantiated from. This is similar to the notion of a *context* [14] for some operations.

Due to the duality of both functions, we can say that for any possible model, a succesfully instantiated element needs to conform to the type model it was instantiated from: $\forall i \in [\![TM_{type}]\!]_{inst} : conf_{type}(i) = true$ Several variations exist for both instantiation [19, 256] and conformance [19, 235, 267]. Nonetheless, the duality of both functions should still be maintained for these functions to retain their value.

It is through these two relations that meta-modelling layers are constructed [169]. A popular architecture for this is the four-layered architecture, as popularized by the Object Management Group (OMG) [4] and shown in Figure 3.2. The top level, M3, defines the Model at the MetaCircular Level (MMCL), which conforms to itself. It is a language for defining metalanguages, and thus general. A popular choice for this level is the Meta-Object Facility (MOF) [4]. The level below, M2, defines the language that will be used to create the user models. A popular choice for this level is the UML [5]. At the M1 level, user models are defined in terms of the language defined at M2. That is, if the M2 level presented a language for UML Class Diagrams, the M1 level will only contain class diagrams (*i.e.*, instances). Finally, the M0 level represents the physical world, and thus the system that was being modelled at M1. In our case, this is the actual system being modelled with the class diagrams.

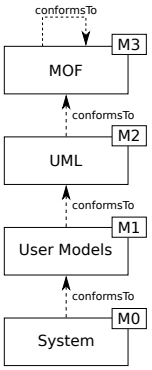


Figure 3.2: Four-layered architecture as popularized by the OMG.

3.1.2 Model Finding and Type Inference

The type/instance relation consists of three components: the model, the type model, and the type mapping. For any combination of these three, it is possible to define whether it is a valid typing relation or not. One cannot be altered without updating the others. It is, however, a common operation to change the model (*i.e.*, normal modelling), the metamodel (*i.e.*, update the language), or even the type mapping (*i.e.*, retype). Due to this strong relation between the three of them, even a minor change requires an update to the other two, which is tedious. For this reason, automated ways exist of automatically filling in the model, metamodel, or type mapping. We present the use cases of each of these individually.

The simplest is the automated generation of a conforming model, which is often called satisfiability analysis [130, 256]. For a given metamodel, instances are generated, which are subsequently checked by the conformance relation. In case of satisfiability analysis, this allows users to find out whether or not their type model is actually satisfiable (*i.e.*, has possible instances). If no instances are found, it means that the type model is overly restrictive. If some instances are found, these can be used as example models, or to prove that instances are possible. Potentially, partial models can be passed along to the search [256], which the algorithm would then attempt to auto-complete, thus creating conforming models [27, 227]. In case of multiple conforming models, auto-completion would require additional information on which one to take.

Another frequently used operation is the generation of a type model to conform to [244]. Exploratory design, or freehand sketching [148], might happen without the advance creation of a type model, with only simple figures representing the model. Similarly, the type model might be overly generic and not really constrain the model too much.

Finally, the type mapping between the model and type model is the third part that is required. While this might seem a less delicate operation than the previous two, it is the mapping between the two that gives an actual meaning to both constructs. An incomplete type mapping can arise when the metamodel or model has been updated without also updating the other. These situations are commonplace in language evolution [197, 266], where either can evolve independently. In general, a pre-existing type mapping can be used as a starting point, but in the limit, this approach would also work if no initial type mapping is offered.

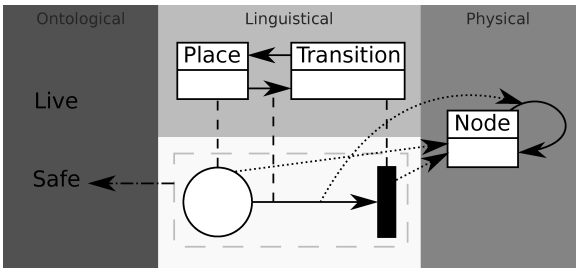


Figure 3.3: Three different classification dimensions: ontological, linguistcal, and physical. All dimensions are orthogonal to each other.

3.1.3 Multiple Dimensions

Up to now, we have only taken into account a single dimension of conformance. While it seems logical that a model conforms to its metamodel, it can do so in multiple dimensions simultaneously [31, 137]. We distinguish three different dimensions: physical, linguistic, and ontological. Our terminology is similar to that of [35, 295, 322], but in conflict with the one originally defined in the Orthogonal Classification Architecture (OCA) [32]. Each of these relations is discussed next. An overview is shown in Figure 3.3, where all three dimensions for the same model are visualized.

The physical conformance relation defines how a model is physically represented in the tool. For example, any kind of model (*i.e.*, a Petri Net place, a Statechart state, ...) will, in the tool, be represented as an object in some programming language (e.g., Python, Java). This classification has nothing to do with the domain, and is purely geared towards the implementation. As the physical conformance dimension is part of the tool, its type model is hardcoded and cannot be modified. It is a necessary dimension, as each element of the model needs to be physically persisted one way or the other.

The linguistic conformance relation defines the type model of the model elements as we have seen in previous sections. For example, a Petri Net place instance conforms to the Petri Net Place class in the type model. Contrary to the physical conformance relation, the metamodel is yet another model, which simply acted as a type model in this specific context. Both the model and type model are therefore modifiable.

The ontological conformance relation defines the properties which the model fulfils. While this is seemingly different, as there is no longer a notion of structure, it is still said that a model can conform to a property. For example, a Petri Net model can conform to the *safe* property. Contrary to the previous conformance relations, ontological conformance takes into account the semantics of the model, instead of only the structure.

For each dimensions, it is possible for there to be multiple type models simultaneously. In the physical dimension, a model can conform to two different type models in case of different representations for the same model. In the linguistic dimension, a model can conform to two different type models if one is more generic than the other, in which case this helps reusability [83]. In the ontological dimension, it is normal that a model can conform to multiple distinct properties, such as safeness and deadlocking [322].

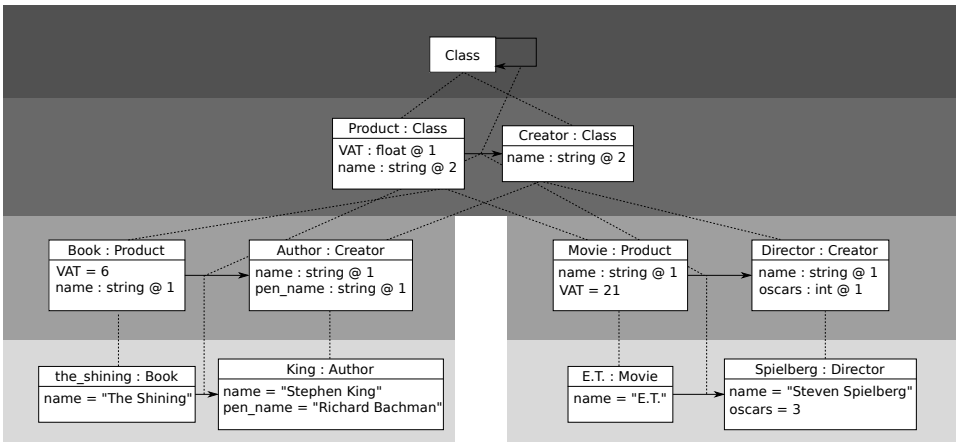


Figure 3.4: Multi-level modelling.

3.1.4 Multi-Level Modelling

The four-layered architecture is a popular modelling architecture, but not the only one. Of the four layers of the four-layered architecture, only two are modifiable (M2 and M1), which can be seen as overly restrictive. Multi-level approaches [33] allow the number of layers to grow unbounded. Despite not being as commonplace as the four-layered architecture [82], multi-level modelling is particularly useful when normally the type/object pattern [33, 126] would be applied, to handle dynamic types [81], or for the execution of models [26]. While there are several other potential solutions to these problems, such as powertypes [212], multi-level modelling is considered to be more elegant, as there are no layer-crossing inheritance links. Multi-level modelling has proven to be a thriving research direction with many different directions and new concepts [23].

Multi-level modelling popularizes the idea of *deep instantiation*, where an element cannot only be instantiated once, but multiple times. The distinction between classes and objects also becomes blurred, as all (except the top and bottom level) model elements have both an instance and a type aspect. As a result, the merger of the two is called *clabject* (CLAss oBJECT), which replaces the previous classes and objects. Facilities are offered to have elements influence instances several levels down, which is called *deep characterization*. Through the use of potency on clabjects [31] and associations [25], restrictions can be made that will propagate several instantiation layers down.

An example is shown in Figure 3.4. Here, we notice that there are three user-modifiable levels, instead of two. This is useful to define the notion of a *Product* and *Creator*, which is recurring in several domains, such as books (left hand side) and movies (right hand side). These different domains all have a notion of VAT and name. The VAT, however, is strictly related to the domain (e.g., books), and must therefore be defined for the domain itself. This is done through the potency (the `@ 1` notation), meaning that the attribute must be defined in the layer below. The name, however, dependends not on the domain, but on the instance. It is therefore given a potency of two (the `@ 2` notation), meaning that the attribute must be defined two layers below (i.e., at the instance level). By definining these attributes at the topmost layer, we are certain that all instances have the correct set of attributes.

Strict Metamodelling String metamodelling [20] is of particular interest in the context of multi-level modelling, where it states that only the conformance relation can cross layer boundaries. Levels can thus be inferred [169], with each level being completely specified by only looking at the type model one level above. Despite its elegance, it is claimed to be too strict for several applications [66, 230], in particular for enactment [138], and with powertypes [82]. Not applying strict metamodelling is called loose metamodelling [138], where there is no constraint whatsoever on the levels at which models reside, and is implemented in XMF-Mosaic [66]. While solving the problems of strict metamodelling, all advantages of strict metamodelling, in particular its strong notion of levels [23], are gone as well.

Summary

In summary, language engineering, and in particular the more theoretical notions of abstract syntax, is an active research area. We focussed on the instantiation and conformance relation, which are the most important relations between models: when does a model conform, and what does it mean if this relation holds? Subsequently, we briefly presented alternative ways of determining such conformance relations, either by automatically finding instance models or by infering the type of instances. Further dimension to conformance were presented in the context of the Orthogonal Classification Architecture (OCA), which describes three different dimensions: physical, linguistic, and ontological. We briefly mentioned the growing domain of multi-level modelling, where the meta-hierarchy is no longer restricted to four layers, and the controversy over strict metamodelling. Note that language engineering encompasses far more than only abstract syntax, such as concrete syntax and the usability of languages. Nonetheless, only abstract syntax is of interest for the remainder of this thesis.

3.2 Activities

Different types of activities exist, for each of which there have been several research directions. In this thesis, we consider the two most popular approaches: model transformations and procedural code.

3.2.1 Model Transformations

Research on model transformations is often focussed on different dimensions of their usability.

The first dimension is making model transformations easier to define. Examples in this direction are through the use of familiar concrete syntax (e.g., through RAMification of the language [171] or by reusing existing modelling environments [9]). Model transformation by example [153] is an alternative way of specification, where the transformations are not specified directly.

A second dimension is the reusability of model transformations. One example is intent-based reuse [248] focusses on the intent of the transformation, instead of on its typing model. Another is the use of typing requirement models [78], which are models that explicitly describe the requirements that the transformation needs from the source and

target metamodels in order to obtain a transformation with a syntactically correct type. Yet another approach focusses on structural heterogeneities, which can be automatically resolved through a flexible binding [332].

A third dimension is the efficiency of model transformations. Novel techniques are introduced such as activity (using locality and domain-knowledge) [92, 151], incrementality (storing and reusing previous matches) [105, 274, 282], and model-sensitive search plans (optimize search plans based on expected performance on typical instances) [324].

3.2.2 Procedural Code

Procedural action code is another popular way of defining model activities. Such languages look similar to existing programming languages, and are indeed often strongly related. This has advantages, such as integrating Java code directly into models, but also disadvantages, such as everything being Java, thus offsetting many benefits of modelling.

Modelling-specific procedural languages have native constructs for common model management operations, such as getting all instances of a class. Examples of such languages are OCL [3], EOL [164], and Kermeta [201]. OCL was designed as a side-effect-free functional constraint language, allowing efficient implementation techniques (e.g., lazy evaluation [280]). EOL was designed as a model management language, targetted at manipulating models and their relations [164]. Several related languages were created, based on the Epsilon platform, such as for model comparison (ECL [163]), model merging (EML [165]), and code generation (EGL [238]). Kermeta was designed as an executable statically-typed object-oriented meta-language [201]. Its semantics was (partially) formally described [16], and later on concurrency was added [174].

In the domain of multi-level modelling, deep constraint languages [24] (e.g., DeepOCL [152]) have emerged. Such languages have built-in notions of potency and deep meta-models, but can also distinguish between linguistic and physical attributes (e.g., the `name` attribute can specify the internal physical "name" attribute, or the domain-specific attribute "name").

Summary

We distinguished two types of activities: model transformation and procedural code. Model transformation is often touted as the more user-friendly approach, though further research is needed to make them more generally applicable (e.g., for transformation libraries), make them easier to define, and to increase their performance. Procedural code does not encounter these problems, as it is a closer map to the actual code implementing the execution. Nonetheless, procedural code is mostly targetted towards programmers and might therefore not be as intuitive for domain experts.

3.3 Processes

Process models describe the often complex workflows of today's systems. Process models can be defined as follows:

Process models are processes of the same nature that are classified together into a model. Thus, a process model is a description of a process at the type level. Since the process

model is at the type level, a process is an instantiation of it. The same process model is used repeatedly for the development of many applications and thus, has many instantiations. [234]

Process models define how domain-specific models are used and which activities operate on them. Depending on the formalism, the process can be analysed or enacted [214]. Through enactment, the engineering services are orchestrated, enabling a higher level of automation. Optionally, activities can be annotated with information on their execution, such as computation time, which domain experts must perform the manual activity, and so on. Apart from control flow, specifying which activities are to be executed, data flow is also modelled explicitly, specifying which models the activities operate on. Activities are either manual or automated (e.g., model transformation or procedural code), and can optionally rely on external (engineering) tools as well.

Being an active research domain, many process modelling languages have been defined for software development by the process engineering community. A notable examples is OMG's Software Process Engineering Metamodel (SPEM) [1], which is a generic framework for expressing processes with generic work items. UML4SPM [37] further adding execution support to SPEM2 with concepts and behavioural semantics. Similarly, Chou et al. [64] present a language based on UML class diagram and activity diagrams, which they map to a low-level process. A comparison of several process modelling approaches and languages can be found in [38] and [139]. Similarly, Business Process Model and Notation (BPMN) [2] is a standard by the OMG focussed on business process modelling, which provides a graphical language representing a flowchart. Its primary goal is to provide a syntax that is intuitive for business users, while still representing sufficient technical details for the technical users.

In the remainder of this thesis, UML2.0 Activity Diagrams are mostly used, which have been evaluated as a process modelling technique [241]. The conclusion was that there was very good support for control- and data flow modelling, though there were lots of limitations for resource- and organiation related aspects. Resource- and organizational aspects of the language are less important for our purposes, and as such we have used (a subset of) Activity Diagrams as process model in this thesis.

3.4 Megamodelling

Megamodelling was originally introduced by Bézivin et al. as "modeling in the large, establishing and using global relationships and metadata on the basic macroscopic entities (mainly models and metamodels), ignoring the internal details of these global entities" [42]. Several definitions can be found in the literature [41], though they all mostly mean the same: a megamodel is a model describing the relations of other models. The FTG+PM [186] can also be seen as a megamodel, as the formalisms and objects it refers to are other models that exist in the tool.

Megamodels have a variety of uses, such as to store meta-data (e.g., inconsistency information [76] and dependencies [225]) or to be used for model management (e.g., develop complex transformations on a global level [231]). For example, MMINT (formerly MMTF [243]) is a model management framework based on megamodelling [100], with support for operations such as differencing, slicing, matching, and merging [246]. MMINT

explicitly stores metadata in the megamodel, using it as a registry for models and metamodels [44].

3.5 Modelling as a Service

With the increasing number of (meta-)modelling tools, the need for model repositories as central model stores is becoming more and more evident [43, 172]. Model repositories are a necessity with collaboration, where multiple (geographically distributed) users collaborate on a single model. This naturally leads to Modelling as a Service (MaaS) [97], where models and all activities on them are offered by a server, to which multiple clients connect. Such software is commonly referred to as groupware [118], and has many concerns (e.g., network management, synchronization, awareness, and shared workspaces), many of which are active research domains in the context of modelling.

Another reason to rely on model repositories, and MaaS in general, is that models become too large for a single system to handle. Many modelling tools are more and more evolving towards a thin-client setup [125, 273], where all computation is done on the server. For example, AToM^3 [86] has evolved to AToMPM [273], GME [175] has evolved to WebGME [192], and DPF [173] also has a WebDPF [228] variant.

3.6 Tool Comparison

After explaining and motivating the goals of this thesis, it is important to highlight that currently no foundation for MPM exists in the literature. We evaluate several relevant tools for their support for these research domains that make up Multi-Paradigm Modelling, as desired in this thesis. As MPM combines different areas of research, the set of considered tools is necessarily rather diverse. While many tools exist in each individual domain, only several were considered here due to their relevance for some particular aspect of MPM. Although not all tools were considered, to the best of our knowledge, no tool exists that combines all these features.

Most of the mentioned tools are prototypes in one of the aforementioned research domains, and are therefore highly specialized. Table 3.1 compares the tools, based on our experiences with and knowledge of each tool. Note that this is necessarily a snapshot, and that tools might add new features in the future. For example, AnyLogic added modelling as a service several months before this thesis was written. While this specific example was included, it is very well possible that we are unaware of specific features in tools. Nonetheless, this does not invalidate our claim that none of these tools support *all* of the desired features, as all considered tools are missing out on multiple features. Further detailed information on each tool is presented next.

AnyLogic AnyLogic [6] is a modelling tool with a primary focus on simulation (i.e., activities). It does not support language engineering, as it is not possible to create new languages: only pre-defined languages can be used. Three simulation languages are supported: discrete event, agent-based, and system dynamics. For these three languages, built-in activities are defined for their simulation. There is no support for processes, nor for megamodelling. Modelling as a service became available in the latest release of AnyLogic

	language engineering	activities	process modelling	mega-modelling	modelling as a service
AnyLogic [6]	○	●	○	○	◐
AToMPM [273]	●	●	◐	○	◐
BPMN [2]	○	○	●	○	○
Groove [232]	◐	●	○	○	○
MDEForge [36]	◐	●	○	◐	●
MetaDepth [79]	●	●	○	○	○
MetaEdit+ [154]	●	○	○	○	●
MMINT [100]	◐	◐	○	●	○
ReMoDD [112]	○	○	○	○	◐
WebGME [192]	◐	○	○	○	●

Table 3.1: Tool comparison as to which domains they support.

with AnyLogicCloud [7]. However, support is limited to sharing models and simulating them.

AToMPM AToMPM [273], or A Tool for Multi-Paradigm Modelling, is a language workbench with support for many features of MPM. There is full support for language engineering (using Class Diagrams, though no multi-level modelling) and activities (only model transformations using RAMification [171]). Other aspects of MPM are only partially supported or not at all. Process modelling is supported, through a provided FTG+PM language. And although enactment support is provided using a separate plugin, it is was not well integrated in the design of the tool. Modelling as a Service is partially supported, as most computations happen on the server, which is also where the models are stored. Nonetheless, only the AToMPM browser interface can connect to it, as the communication protocol and model representation format is non-trivial and undocumented. Collaboration is supported through model share (share abstract syntax) and screen share (share concrete syntax as well), although there is no support for user access control management.

BPMN BPMN [2], or Business Process Model and Notation, is a standard for graphical notation for the modelling of business processes and is a standard by the Object Management Group (OMG). While BPMN denotes a standard, and not a tool, the standard only relates to the modelling of processes. Apart from the modelling of processes, and its potential enactment by a tool, BPMN does not touch upon any of the other dimensions of MPM. It was, however, included in our comparison, as it is arguably one of the most influential process modelling languages.

Groove Groove [232] is a graph transformation language and tool. As the description suggests, most attention is oriented towards activities: graph transformation execution and state space analysis. Some aspects of language engineering are present, such as defining a

grammar, but this is rather restricted compared to more specialised language engineering tools (e.g., no concrete syntax, no multi-level modelling). It is possible to define (complex) priorities for the applications of transformation rules, though this is far from actual process modelling and enactment. All other aspects of MPM (megamodelling and MaaS) are untouched.

MetaDepth MetaDepth [79] is a meta-modelling tool that primarily focusses on language engineering, and support for multi-level modelling in particular. It provides extensive support for language engineering (such as customisable textual concrete syntax [84]) and to a lesser degree also for activities (primarily action language and model transformations [79]). Other aspects, such as process modelling, megamodelling, and modelling as a service, are untouched.

MetaEdit+ MetaEdit+ [154] is a commercial domain-specific meta-modelling environment. MetaEdit+ is mostly focussed on language engineering, and using these languages to create new domain-specific models. No support is provided for activities on these models, neither through model transformation nor procedural action languages. Nonetheless, a plugin was made to allow for model transformation, through the use of RAMification [301], though the execution then runs outside of MetaEdit+. No support for processes or megamodels is present either. While MetaEdit+ is a commercial, proprietary tool, it does implement a SOAP API with which external tools can query and modify the models stored in the tool. Recently, support was added for model repositories, which can be used to do modelling as a service, enabling collaboration.

MMINT MMINT [100], or Model Management INTeractive, is a model management tool based on the Eclipse framework, and was formerly termed the Model Management Tool Framework (MMTF) [243]. As the name suggest, it is mostly oriented towards model management and megamodelling. In this domain, it provides collection-based operators to manipulate an entire graph of related models. Other aspects, such as language engineering and activities, are of lesser importance here, though some are supported due to its integration into the Eclipse framework. No attention is payed to aspects such as process modelling and modelling as a service, although it is stated that they assume the presence of a model repository.

MDEForge MDEForge [36] is primarily a model repository, which also supports modelling as a service. MDEForge is based on top of the Ecore standard [8], and accepts models in that representation. While some modelling operations can be done using the underlying Ecore library, language engineering is not the core focus of MDEForge. It is primarily intended as a model repository with MaaS capabilities, thereby offering basic language engineering and activity operations. For example, existing models can be transformed to other models through the web interface. Since recently, ATL [149] and ETL [166] model transformations can be uploaded and executed as well [99]. There is no support for the creation, nor the enactment of processes, nor is there support for megamodels.

ReMoDD ReMoDD [112] is a model repository that is designed to host a wide variety of models. Given the difficulties on storing models in a variety of representations, ReMoDD has chosen to lean towards genericity, thereby allowing everything to be uploaded. While indeed everything can be seen as a model, this generic approach makes it impossible to operate on the uploaded models: there is simply no way to know how to operate on the models. Additionally, only a minimal amount of meta-data is available to process or filter upon, meaning that it is solely a model repository without much operations on it. Apart from a model repository, there is no support for any other dimension of MPM

WebGME WebGME [192] is an online collaborative language workbench. It mostly focusses on collaborative modelling, based on model versioning, and it can therefore also serve as a model repository. While it can be used for language engineering, there are several limitations (e.g., no custom concrete syntax) which limit its applicability. Noteworthy, it relies on prototypical inheritance, making it stand out from the rest of the tools. Activities are only minimally supported, and often only through the use of external plugins. There is no support for process modelling, nor for megamodelling.

Summary

MPM spans several research domains, all of which have to be incorporated in our tool to some extent. We described recent advances in language engineering (focussing on conformance and multi-level modelling), activities (both model transformations and procedural code), processes (and their enactment and analysis), megamodels (with applications for model management in particular), and modelling as a service (with links to collaborative modelling). It is not our ambition to implement the state of the art of each of these research domains in our prototype tool, though it provides the necessary context in which this thesis is to be placed. Our prototype should be adaptable to handle new developments, at least conceptually. The provided background in each research domain provides additional insights in the domain, which are referred to throughout this thesis. We compared the support of several tools for each of these domains. Unsurprisingly, these tools excel in the research domain they target (i.e., they implement most of the state of the art), though they (mostly) ignore other relevant domains to MPM.

Chapter 4

Modelverse Specification

One of the core contributions of this thesis is the creation of a foundation for MPM, combined with a tool implementation. In this chapter, we describe the various users and requirements for a foundation for MPM, working in a top-down fashion. Then, a specification is given for the implementation of the various components that make up our tool implementation. We term this specification and its reference tool implementation "the Modelverse".

4.1 Types of Users

We consider there to be three different types of users, for which our foundation is mainly designed. For each user, we consider their classification by mentioning their background and responsibilities. Figure 4.1 presents the meta-hierarchy for a simple Petri Nets example, highlighting the relevant level(s) for each users.

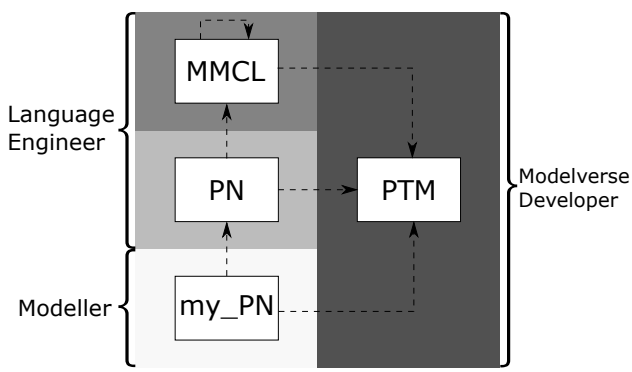


Figure 4.1: Three types of users in relation to the meta-hierarchy.

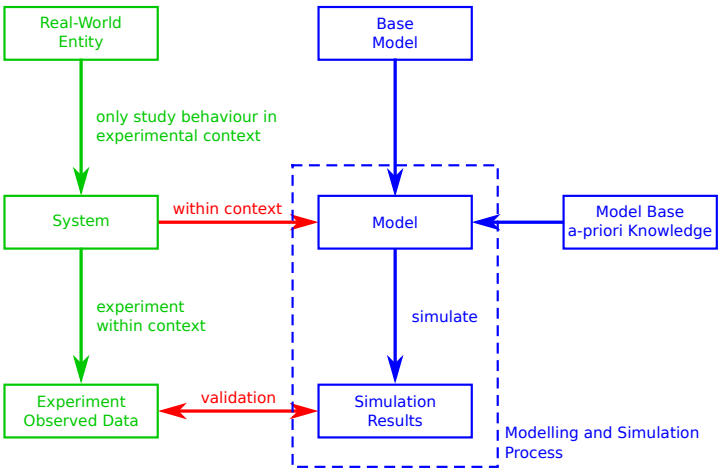


Figure 4.2: The relation between the model and the system. [337]

4.1.1 Modeller

The first type of user is the *modeller*, representing the majority of users. The modeller creates a model of a system under study, and has a background in this domain, making him a domain-expert. Nonetheless, the modeller generally has no or minimal experience in computer science and programming. Typical modellers are engineers, who model the system to achieve additional insights in, for example, the behaviour of the system.

Responsibilities The core responsibility of the modeller is to create an accurate model of the system in question. As modelling requires abstraction [169], it is important to consider the correctness of the model regarding the properties in question. Figure 4.2 shows the relation between a model and the system, given some property. In essence, both the model and the system must return the same response when asked whether a property holds or not. This is the responsibility of the modeller, who created the model from the system.

Dependencies As the modeller is expected to accurately model the system, this job relies on good language and tool support. Given that the modeller has no background in computer science, creating the language naturally falls outside of the modeller's responsibilities. Good language support can be described as being clear and concise, having no unnecessary or missing concepts, having semantics defined, and so on. Tool support is necessary to allow the model to be created and manipulated (partially) automatically.

4.1.2 Language Engineer

The second type of user is the *language engineer*. Language engineers design and create languages used by modellers to model the system. They work at the meta-level, where they constrain the modellers as to which concepts they can use and how they can use them. A background in language engineering is required, as well as general knowledge of the problem domain and computer science. Knowledge of the problem domain allows them

to meaningfully construct a language that can be used in the domain, which is not overly restrictive or has missing concepts, and to know what is the expected semantics. Knowledge of computer science and language engineering is required to create the language, which often involves some fragments of general purpose programming languages, for example to define constraints over the language or to define operational semantics.

Responsibilities The core responsibility of the language engineer is to create usable languages, which are intuitive for the modeller to use. The language engineer thus works in function of the modeller, and has to ensure that the modeller can use this language to correctly model the system. This requires the definition of abstract and concrete syntax, but semantics as well. Abstract syntax design requires an analysis of the problem domain to reveal the various concepts and their relations, though sketch-based approaches also exist, which construct the abstract syntax from a set of example instances [183]. Concrete syntax design can use existing guidelines [198], though will require significant evaluation. Semantics must also be defined, (efficiently) implementing the meaning of the language as it is also understood by the modellers.

Dependencies Just like the modeller, the language engineer depends on tool support for all its responsibilities. In contrast to the modeller, however, the language engineer requires methods to create new languages, which can be loaded and used by the modellers. To define the semantics, language engineers depend on tool support for different specification languages. Language engineers also depend on the existence of a meta-language, although this is essentially also specified by (another) language engineer.

4.1.3 Modelverse Developer

The final group of users considered are the developers of the MPM tool itself. While not users in the same sense as the modeller and the language engineer, they are also required to use their own tool. They are in the minority, as only a select group of people will have to develop the tool. Modelverse developers have a strong background in computer science, particularly in programming, as they have to know all about the nuts and bolts of the tool.

Responsibilities The core responsibility of tool developers is to ensure a bug-free environment that fulfills the needs of the other users of the tool (i.e., the modeller and the language engineer).

Dependencies The tool developer depends on the lower level language that was used for the implementation of the tool, such as C++ or Python. These languages and their tools, such as interpreters, compilers, and debuggers, are all used by the tool developers. In the context of this thesis, we will not go deeper into this.

Summary

We considered three types of users for a foundation for Multi-Paradigm Modelling. The modeller models an existing system using a provided language, aiming to develop a correct

model with respect to some properties. The language engineer creates the languages and semantics for the modeller to use, aiming to make it maximally usable for the modellers. The Modelverse developer creates all required tooling and libraries for all other users to use, aiming to do so without bugs and efficiently.

4.2 Requirements

A foundation for MPM must handle all aspects of MPM, both explicitly mentioned in the definition and implicitly caused by the implications of these explicit requirements. Due to the implications of MPM (e.g., having multiple domain experts, each possibly using a domain-specific tool), it becomes necessary to design the tool as a kernel for model management. Additionally, the presence of multiple users naturally implies its nature as a model repository. The actual tool can therefore be described as a Multi-Paradigm Modelling kernel and repository. We work top-down in listing our requirements, starting from the need for an MPM tool, going to a set of high-level requirements. In the remainder of this thesis, we link back to these requirements and specify how they were satisfied.

For each requirement, we put it in the context of an example: the power window case study The power window example consists of a simple electronically controlled window of a car. Users control a button (the *environment*), which can be in three modes: up, down, or neutral. This button is connected to a controller (the *control*), which translates the keypresses into commands to the engine responsible for window movement. The window itself is raised by a wormgear (the *plant*), which can also be in three modes: up, down, or neutral. While there is an intuitive mapping between the controls and the window behaviour (*e.g.*, when the up button is pressed, the window should go up), there are some corner cases or additional requirements. In the context of MPM, a power window is often used, as it is relatively minimal and easy to understand, while still posing many of the challenges faced in more general MPM problems [91, 200]. We focus on the verification aspect of the power window: we want to make sure that when an object is inserted through the window, the window will never exert a high power (*i.e.*, keep going up).

While we cannot guarantee that this list of requirements is complete, we later on evaluate the Power Window case study on a tool implementing all these requirements.

4.2.1 Multi-Paradigm Modelling

The first few requirements considered are those that are explicit from the definition of MPM. MPM combines at least three research domains [316]: language engineering, activities, and process modelling, each resulting in a requirement.

Language Engineering

The first requirement considered, is language engineering. Specifically, this means that domain-specific languages and models can be created and manipulated, such that modellers can make use of the most appropriate formalisms.

Requirement 1: New domain-specific languages and models in these languages must be creatable.

Why? This requirement is explicitly present in the definition of MPM, which requires *"the most appropriate formalism(s)"*. As the most appropriate formalism depends on the domain, and is likely specific to it, this naturally implies the need for Domain-Specific Modelling (DSM) and Domain-Specific Modelling Languages (DSML) support.

Power Window For the power window case study, language engineering makes it possible for each type of modeller to use a domain-specific formalism. Indeed, the plant and controller are modelled by a plant engineering and control engineer, respectively. Both types of engineer have a different background and will want to use a different representation. Otherwise, they all would have to resort to creating low-level models directly, which is far from their area of expertise and the problem domain. Additionally, different goals require different formalisms to be used. For example, for analysis all modellers would have to create a model in the Petri Nets formalism, and then for simulation they would have to create a different model in the CBD formalism. Not only are the modellers likely unfamiliar with those formalisms, but there is also no guarantee that both models represent the same system, as there is no relation between the Petri Nets and CBD model.

Activities

The second requirement considered, is modelling and execution support for activities. Specifically, it must be possible to define an activity that takes a set of models as input, and returns another set of models as output. An activity might thus create new models or delete existing models, as is required when translating from one formalism to another, or might update a model in place, as is required when simulating a model. There is a distinction between manual and automatic activities: where automatic activities execute completely automatically (i.e., loading the models, performing the activity, writing out the changes), manual activities require some kind of user intervention. Even for manual activities, all required input models are loaded automatically, just like the languages that can be used to model with.

Requirement 2: Activities must be specifiable and executable in many different formalisms.

Why? This requirement stems from the need for *the most appropriate formalism(s)* in the definition of MPM. Indeed, the use of domain-specific languages can only come to full fruition if these models have a semantics, which is given through activities. Additionally, the use of the *most appropriate formalism(s)* also reflects on the activity specification. For example, when constructing the reachability graph of a Petri Net model, this is ideally done in an operational language, as several algorithms for that exist already. Conversely, when translating between two languages, this might be ideally done using a declarative model transformation.

Power Window For the power window case study, this implies the need for activities to map between different formalisms. On one hand, we want to define activities that map models in the domain-specific formalisms to models in a general-purpose formalism (e.g., Petri Nets), thereby implementing denotational semantics (through declarative model

transformations). On the other hand, we also want to define activities on these general-purpose formalisms to give them semantics, such as computing a reachability graph (through operational action code).

Processes

The third requirement considered, is process modelling and enactment support. Specifically, it must be possible to model a process in a process modelling language, such as Activity Diagrams or BPMN [2], taking into account control flow and data flow. This process can then be automatically enacted by executing the activities (manual or automated) on the correct models, in the correct order, potentially concurrently.

Requirement 3: Process models can be created and enacted using previously defined models.

Why? This requirement stems from the need for an *explicitly modelled process* in the definition of MPM. This was already implied in the *model all relevant aspects*, even though the process is indeed not part of the built system, but was made explicit in later revisions of the definition. To maximise the usefulness of the process model, enactment support should also be provided.

Power Window For the power window case study, this implies the need for an explicitly modelled process, as is given in the FTG+PM. This FTG+PM can then automatically be enacted, thereby executing activities automatically. Indeed, this makes it that we merely have to enact the FTG+PM termed "verify power window", after which the tool automatically prompts the various users in the correct order, hiding all lower-level languages (such as Petri Nets) from the modellers. Manual activities still require modeller intervention, though the correct languages and models are already pre-loaded where possible. Automated activities happen in the background, and possibly concurrent to other activities, completely invisible to the modellers. For the previously defined FTG+PM (Figure 2.22), this means that not a single modeller is exposed to the intermediate Petri Nets and corresponding reachability graph.

4.2.2 Kernel

The second aspect of the foundation for MPM that we will consider, is its use as an MPM kernel. An MPM kernel is characterized as a service (Modelling as a Service [97]) that coordinates the interaction of different individual entities, particularly in their use of shared resources. For an operating system, these entities are processes and individual threads. For an MPM kernel, these entities are different clients (user interfaces, responding to user input), different activities spawned by these clients, and different external services. All these entities have to share the limited resources available at the server, such as computation power, storage, and network. MPM implicitly raises the need for a kernel, as it requires multiple domain experts to interact and cooperate with one another.

Multi-User

The fourth requirement considered, is support for multiple users. There can be multiple such users, all connecting simultaneously from geographically distributed locations. Users don't interact directly with the MPM kernel, but should be guided by a (user-friendly) interface. Through these interfaces, users are able to perform Modelling as a Service operations, such as creating new models, removing models, executing activities, enacting processes, and so on. While doing all this, fairness should be guaranteed, such that, for example, no single user can hog all computational resources or memory. This is particularly important with the execution of activities, which can take a decent amount of time (e.g., long-running simulation) and memory (e.g., state space analysis).

Requirement 4: Multiple users must be allowed to use the same tool simultaneously, potentially from geographically distributed locations, equally sharing resources such as storage and computation.

Why? Modelling all aspects of the system explicitly using the most appropriate formalism(s) naturally implies that domain experts will model themselves, instead of having single all-purpose modeller. To optimize the process, it is only natural that multiple such modellers are operating concurrently, meaning that multiple domain experts (from different domains) work concurrently. As all these types of modellers are equal, no single modeller should ever occupy all shared resources.

Power Window For the power window case study, this makes it possible that the various domain-specific modelling operations (e.g., model the plant and the controller) can occur concurrently by different users (e.g., a plant engineer and a control engineer). Indeed, in our example FTG+PM (Figure 2.22), no less than five domain-specific models were being created concurrently, all by potentially different users. In this context, it is only natural that for example the plant engineer can not hog all computation resources, causing the control engineer to be stuck.

Multi-Service

The fifth requirement considered, is support for multiple external services. Specifically, activities can be defined as being "external" automatic, meaning that they interact with some external tool, implementing (parts of) the desired functionality. No constraints are placed on these external tools, as they might be oblivious of MPM and even modelling in general. Additionally, no constraints should be placed on implementation restrictions of such tools: they might even run on a different platform, and should be completely black-box to our foundation for MPM. It should be possible to connect to such external tools and use them in a (preferably) automated way.

Requirement 5: Multiple external (proprietary) services must be able to connect and operate concurrently, potentially from geographically distributed locations.

Why? This requirement stems from the support for various domains, many of which have their own domain-specific tools and solvers. While it is technically possible to reimplement all such features in a single tool, this is not be a viable option. Many of these algorithms

have taken many man years to develop, resulting in advanced features and high performance. It is therefore important to reuse existing tools and solvers as much as possible, instead of reinventing the wheel.

Power Window For the power window case study, this implies that external tools can be used to perform some activities. For example, the reachability analysis on the Petri Net model is non-trivial to re-implement, and certainly not if performance is a concern. Luckily, many efficient implementations exist in dedicated tools, such as LoLa [254]. Through the use of external services, it becomes possible to define an activity termed "analyse reachability", which interacts with LoLa instead of doing the analysis itself.

Multi-Interface

The sixth requirement considered, is support for multiple user interfaces, possibly implemented for different visualization formats and implementation platforms, using different programming languages and libraries. This merely requires that client and server should not be overly coupled, making it possible to switch out any of them for another implementation. This results in the need for a simple and intuitive interface, as well as some unified data representation. Nonetheless, this data representation should not make assumptions of the client, as there might exist a visual client (with notions of symbols and figures) and a textual client, both of which should be able to offer the same functionality. As such, this requirement goes beyond purely technological restrictions.

Requirement 6: Multiple different user interfaces must be able to connect simultaneously, each one possibly developed with different languages and libraries.

Why? This requirement stems from the support for various domains and users with different backgrounds: each user has his/her prefered tool and type of interface. There is for example the distinction between textual and visual interfaces, with some research on figuring out which is the "best" format [128, 219]. Nonetheless, as reflected in the definition of MPM, we consider that there is at most a "most appropriate" format, depending on the domain and user. For example, while visualization is most appropriate for architectural plans, textual representations might be better suited for a procedural action language. Similarly, domain experts might want to stick to the tools they are familiar with, making it necessary that these tools can be extended to communicate with our MPM kernel.

Power Window For the power window case study, this implies the need for multiple types of interfaces, possibly for the different types of users. Depending on domain expert preference, the visual or textual representation of the models under consideration can be considered. For example, a textual representation is appropriate for the requirements model, whereas visual models might be more appropriate for a control model. Similarly, tool preference depends on the background of individual users: the plant engineer is likely familiar with other tools than the control engineer.

4.2.3 Repository

Another set of requirements comes from the need for repository functionality. Due to the nature of MPM, where multiple users are collaborating on a single system, there is a need to share modelling artefacts, such as models, metamodels, and activities. While Modelling as a Service considered computation, a repository considers data. Indeed, one can occur without the other, as seen in for example ReMoDD, which acts as a repository, but does not support MaaS. The repository aspect of our tool considers three additional requirements that should be imposed: enabling sharing of data, permissions, and managing the data.

Share Models

The seventh requirement considered, is support for model sharing. Specifically, models created by one user might be visible to other users as well, such that they can interact (e.g., one user uses the models of the other as input). While we state that models can be shared, this naturally extends to languages and activities as well, as all of these are to be represented as models in an MPM context.

Requirement 7: Models (in a broad sense) must be sharable between users.

Why? This requirement stems from the various users who need to collaborate (see **Requirement 4 (Multi-user)**), thereby needing to share models. Additionally, when enacting a process, all involved users must be able to access the used models, languages, and activities that are assigned to them. This requirement is therefore tightly interleaved with many of the other requirements.

Power Window For the power window case study, this implies that the different types of modellers must have access to their own models (e.g., the plant engineer must be able to access the plant models). While initially the models are created from scratch, it might be that a different plant engineer is called in to do the second iteration (e.g., refinement). Similarly, the activity merging the different Petri Net models should have access to all models, but also to the semantics of this merge, which is stored as a model as well. Many of the languages and activities involved in this process, such as (Encapsulated) Petri Nets and their analysis, are reusable for other projects as well.

Access Control

The eighth requirement considered, is support for access control. Specifically, while models can be shared among users, they must be augmented with meta-data on permissions, signalling what type of access is provided to other users and groups (no access, read-only, or write). To simplify access control management, this can optionally make use of a group-based access control system, where users can be a member of a group, thereby inheriting all permissions of that group as well. Apart from readability and writeability, the notion of executability is also important: an efficient algorithm might have to be protected from inspection (i.e., made unreadable), although it should still be executable (e.g., by customers).

Requirement 8: User access control must be present to regulate sharing of models (in a broad sense) between users and groups.

Why? This requirement stems from the various modellers, possibly from different companies and with access to different parts of intellectual property. Despite the requirement for model sharing, many organizations do not wish to share their intellectual property with too broad a user base. Such organizations would therefore be relunctant to adopt our foundation for MPM if there were no concept of access control built-in.

Power Window For the power window case study, this makes it possible that the plant engineer only has access to the plant model. Indeed, there is no functional reason as to why a plant engineer must access the control models. In our FTG+PM, it is even possible to make the combination operate on a black box: the intermediate Encapsulated Petri Nets might be the only artefacts readable to the integrator. This maximally restricts access to intellectual property, though of course depends on how easy it is to reverse engineer the domain-specific model from the Petri Net. More generally, such restrictions also apply between different organizations, as all use the same service: plant engineers from company A should not have access to plant models from company B.

Megamodelling

The ninth requirement considered, is support for megamodelling. Specifically, links can be created between models to highlight a specific relation between them, such as model management operations [100] or consistency [75]. Additionally, megamodelling is often used to store meta-data about models, such as their access permissions or traceability links (e.g., automatically generated out of some other model).

Requirement 9: Megamodelling must be supported, allowing for meta-data on models and links between them.

Why? This requirement stems from the various models, languages, and activities used due to the nature of MPM. Many such models have relations to one another, making this additional information useful during the development process: they can be used to find related models [36], maintain consistency, used as information for model management operations, and even for model versioning [53].

Power Window For the power window case study, this implies that relations between the various languages can be stored and used. For example, reachability analysis result need to be mapped back to the domain-specific level, where they utilize the traceability information stored during the various translations. Additionally, the activities that rely on model transformation are instances of a RAMified metamodel, being the merger of all involved metamodels. To ensure consistency, and make it possible to update this language, there is a link from this RAMified metamodel to a merged metamodel, and from this merged metamodel to all involved metamodels.

4.2.4 Non-Functional Requirements

Like all software, there are various non-functional requirements relevant to a foundation for MPM. Nonetheless, most of them are generally applicable to most pieces of software,

such as performance, dependability, and maintainability. As this is not specific to MPM, we mostly ignore such requirements throughout this thesis, as they are out of scope. One non-functional requirement, however, is of particular importance to MPM, and is discussed next: portability.

Portability

The tenth and final requirement considered, is portability. Specifically, all aspects of our foundation, ranging from parts of the tools to the models developed in the tool, are to be fully portable between implementation platforms. We consider implementation platforms in the broad sense, including used programming language, used operating system, and used hardware. In essence, our foundation and its implementations should not be grafted on any aspect that is difficult to port among platforms, such as programming language libraries.

Requirement 10: All aspects of our approach must be fully portable between different implementation platforms.

Why? This requirement stems again from the appropriateness that is all-important in MPM. Many formalisms include a notion of "code fragment", which is some executable text, for example in Python. Binding such code fragments to a specific programming language makes it difficult to provide support for it. On the tool side, this implies that the foundation can only be implemented using Python, and that all implementations must be done in the Python programming language. On the model side, this implies that the tool should have execution support for Python, which hints at a tool implementation in Python. Using a complex general-purpose language therefore has significant repercussions on all aspects of the foundation. As such, we opt that our approach should not be based on a complex general-purpose language.

Power Window For the power window case study, this implies that our complete model is to be independent of the language used to implement the modelling platform (e.g., Python, Java). This includes the procedural action code, which is used to for example compute a state space or merge together different Petri Nets. By writing this in a minimal though expressive language, models become independent of the underlying platform.

Summary

We have defined ten requirements for our Multi-Paradigm Modelling foundation, all of which follow from the definition of MPM and the systems it targets. These requirements can be summarized as follows:

1. Domain-specific languages and models in these languages must be creatable.
2. Activities must be specifiable and executable in many different formalisms.
3. Process models can be created and enacted using previously defined models.
4. Multiple (distributed) users equally share computational resources.

5. Multiple external (proprietary) services must be able to connect.
6. Multiple interfaces must be supported, possibly using a different platform.
7. Models must be sharable between different users.
8. User access control regulate sharing of models between users and groups.
9. Links between models must be representable and can be manipulated.
10. All developed tool components must be fully portable between platforms.

4.3 Architecture

The presented requirements immediately hint at a client-server architecture to be used for our prototype implementation. We term our prototype "the Modelverse", and it will consist of three individual projects. The client-side component is called the *Modelverse Interface* (MvI), providing a user-friendly interface. The server-side component is subdivided in two responsibilities: computation and storage. Computation will be handled by the *Modelverse Kernel* (MvK). Storage will be handled by the *Modelverse State* (MvS).

As the MvS is only responsible for storage, the interfaces communicate exclusively to the MvK, which produces required operations on the storage. The MvI will translate commands by the user to high-level MPM-specific operations (e.g., *"create model"* and *"enact process'*). The MvS merely exposes a minimal graph API (e.g., *"create node"*, *"read edge"*), and can be an existing graph database. The MvK, being the glue between MvI and MvS, is then responsible for translating the high-level operations of the MvI to low-level operations of the MvS.

We now describe the full interface of these three components.

Trade-offs During the design of this architecture, various trade-offs were considered at different levels. For example, different network protocols (e.g., XML/HTTPRequests, WebSockets, ZeroMQ) and different storage options (e.g., In-memory database, RDF) were considered. In the interest of space, however, all these alternatives and their trade-offs are not mentioned here.

4.3.1 Modelverse Interface

The Modelverse Interface (MvI) is any type of interface to the users, which can communicate with the Modelverse. Its primary use is to make the Modelverse as user-friendly as possible to the end-user, as indeed this will be the only component that the user interfaces with directly. In essence, it is any type of program that generates high-level MPM-specific requests to the Modelverse. The Modelverse operates independent of which type of interface is used. Some example interfaces are listed next, all of which were implemented in this thesis.

Graphical User Interface The first type of interface is a Graphical User Interface (GUI), for example implemented in TkInter. With such an interface, users have a graphical depiction of the model they are manipulating using mouse-based interaction. A simple example would be to modify a model element with a middle-click and remove it with a control-right click. When these commands are sent to the GUI, the GUI translates this to a model operation in the Modelverse, after which the Modelverse is contacted to actually perform the operation.

Textual User Interface The second type is a textual user interface, where users specify a model in a textual format, such as HUTN notation. With this type of interface, users have a textual representation of the model in some human-readable notation. This text is subsequently parsed and the appropriate high-level operations are sent to the Modelverse.

Programming Language Interface The third type is a programming language interface, or API, where language bindings are created, for example for Python. This is a simple type of interface, as the high-level operations are merely made available as a function in the desired programming language. Upon invoking the function, the corresponding high-level operation is sent out over the network to the Modelverse, and the return value of the Modelverse is parsed and ultimately returned to Python. The only task of this interface thus consists of serializing the (primitive) parameters and deserializing the return value of the call. Of course, some additional wrapping is necessary, for example to handle different modes the client is in and to handle potential exceptions.

Console Interface The fourth type of interface that we consider, is a console-like interface. This is the simplest type of interface, as it merely provides a protocol-level view of the Modelverse. As Modelverse operations are high-level, they can be invoked directly as well, in a prompt-like way. Such an interface merely requests output from the Modelverse, and has the option to send in some data. No modelling logic whatsoever is encoded in this interface.

4.3.2 Modelverse Kernel

The Modelverse Kernel (MvK) is the computational core of the Modelverse and processes the high-level operations. For each of these operations, semantics is briefly and informally described to give an idea of the interface made available to the MvI. A complete definition of what each operation does, can be seen in the models stored in the Modelverse itself. To process these operations, the stored model(s) need to be modified or queried. All storage, however, happens in the Modelverse State (MvS), and is therefore external to the MvK. The MvK thus generates low-level storage operations for the MvS to fulfill the requested operations. For example, the high-level operation “instantiate_node” requires several low-level operations on the underlying graph: a query is made to check whether this instantiation is allowed (reading several nodes and edges), the new node is created, the typing information is updated (again reading and modifying some nodes and edges), and so on.

Operation	Semantics
upload	Overwrite the existing model with a new model.
instantiate_node	Instantiate a new node.
instantiate_edge	Instantiate a new edge.
attr_add	Set an attribute.
attr_add_code	Set an attribute with code.
attr_del	Delete an attribute.
attr_name	Change the name of the attribute.
attr_type	Change the type of the attribute.
attr_optional	Change the optionality of the attribute.
delete	Delete an element.
nice_list	Fetch a pretty-printed list of elements.
list	Fetch a raw list of elements.
JSON	Fetch a JSON serialized list of elements.
read_outgoing	Read all outgoing edges of an element.
read_incoming	Read all incoming edges of an element.
read	Read an elements details.
read_attrs	Read the list of attributes.
read_defined_attrs	Read the list of attributes defined by this class.
types	Read the list of available types.
retype	Retype an element.
read_association_source	Read the source of an association.
read_association_destination	Read the destination of an association.
connections_between	Read the connections between two elements.
all_instances	Read all instances of an element.
define_attribute	Define a new attribute.
undefine_attribute	Undefine an attribute.

Table 4.1: Modelling operations in the MvK.

Modelling

An overview of the various modelling operations is shown in Table 4.1. All these operations operate at the level of a single model, and allow for basic Create-Read-Update-Delete (CRUD) operations on the model. Some operations, such as `read_defined_attrs`, can be rewritten using other existing operations, but are included nonetheless for usability and efficiency reasons. Indeed, it is much more efficient to execute a loop at the server side, than to do the loop at the client side.

Megamodelling

An overview of all megamodelling operations is shown in Table 4.2. In essence, the Modelverse core data structure is a model itself (i.e., a megamodel, see Section 5.5), and could be manipulated using the previously mentioned modelling operations. However, it would not work well to give users (write) access to the complete megamodel, as this includes passwords and intellectual property. As such, these operations provide privileged access to only some parts of the megamodel (e.g., the user's own folders), while protecting others. For the admin user, who has read/write permissions to the core megamodel, these operations are technically unnecessary. For unprivileged users, these operations provide the only way of altering the megamodel, which is sometimes necessary even for non-privileged users (e.g., create a new model).

Operation	Semantics
model_add	Create a new model.
model_rendered	Read all concrete syntax versions of a model.
model_types	Read all supported metamodels for a model.
verify	Verify the conformance of a model.
model_overwrite	Overwrite an existing model.
model_modify	Modify an existing model.
model_delete	Delete an existing model.
model_list	List all available models in a directory.

Table 4.2: Megamodelling operations in the MvK.

Operation	Semantics
transformation_between	Read all activities between two languages.
model_render	Render the selected model.
transformation_execute	Execute an activity.
transformation_add_MANUAL	Create a new manual activity.
transformation_add_AL	Create a new action language activity.
transformation_add_MT	Create a new model transformation activity.
transformation_read_signature	Read the signature of an activity.

Table 4.3: Activity operations in the MvK.

Activities

An overview of all activity operations is shown in Table 4.3. While to the Modelverse activities are models as well, users will appreciate a more specific interface when creating activities. For example, when creating model transformations, the various metamodels have to be merged and RAMified, all of which happens automatically in these operations. As such, these operations provide all necessary processes to follow when creating or executing activities.

Processes

An overview of all process operations is shown in Table 4.4. These operations are limited to executing and reading a process model, as typed by the *ProcessModel* formalism in the Modelverse. As a process model is just an ordinary model, conforming to the *ProcessModel* metamodel, no additional operations are required. Support for other process modelling languages could be added by either translating them to the aforementioned formalism, or by implementing additional functions directly.

Access Control

An overview of all access control operations is shown in Table 4.5. These operations provide the usual operations for a simple group-based access control policy. As for example

Operation	Semantics
process_execute	Enact a process model.
process_signature	Read the signature of a process.

Table 4.4: Process operations in the MvK.

Operation	Semantics
permission_modify	Modify permissions of a model.
permission_owner	Modify the owner of a model.
permission_group	Modify the owning group of a model.
group_create	Create a new group.
group_delete	Delete a group.
group_owner_add	Add owner to a group.
group_owner_delete	Delete owner of a group.
group_join	Join user to group.
group_kick	Kick user from a group.
group_list	List all groups.
admin_promote	Promote to admin user.
admin_demote	Demote to normal user.
user_password	Change user password.
user_logout	Logout user.

Table 4.5: Access control operations in the MvK.

Operation	Semantics
service_register	Register external code as service.
service_get	Get data from Modelverse.
service_set	Send data to Modelverse.
service_poll	Poll for available data in Modelverse.

Table 4.6: Service operations in the MvK.

in UNIX systems, all models have permissions associated with them as meta-data, which dictates what are the owner permissions, group permissions, and other permissions. Each model is associated with a single owner and a single group. Users can be part of multiple groups. The user that is the owner of the model has the permissions given as *owner permissions*. Users that are a member of the owning group have the permissions given as *group permissions*. All other users have the permissions given as *other permissions*. Each group, additionally, can have a set of admin users, who can manage who is a member of that group. Apart from that, there are some general operations for account management (e.g., administrator privileges and password management).

Services

An overview of all service operations is shown in Table 4.6. These operations provide the necessary interface for external services, which have to communicate with the Modelverse. As services are likely not (meta-)modelling tools, they have little use of the usual (meta-)modelling operations. Therefore, the services have more generic set and get operation, which sends data to a specific activity in the Modelverse, or reads data from a specific activity in the Modelverse. The Modelverse manages these different activities and their communication with the outside world.

4.3.3 Modelverse State

The third and final component of the Modelverse is the Modelverse State (MvS). While the Modelverse Kernel is responsible for the core logic, data storage is the responsibility of the Modelverse State. The MvS maintains a single graph-like data structure, manipulated

through a simple CRUD interface. Contrary to existing tools, which sometimes also offer a graph-like structure, our data structure has no special extensions, such as inheritance [177], containment [177], or conformance [122]. It is only at the Modelverse Kernel (MvK) level that an interpretation is given to this graph and its elements (e.g., as an edge that represents inheritance).

The MvS is the only part of the Modelverse that we did not model explicitly, as it is purely algorithmic and likely makes use of external libraries. Additionally, for performance reasons this will likely become a proprietary graph database, which has to be considered as a black box. As there is no model to refer to, in this case, a precise description of the MvS interface [305] is given in Appendix A. What follows is a high-level explanation of the MvS and its interface.

Data representation

Informally, we define the MvS data structure a graph which can have a single primitive value in a node, and both nodes and edges can be connected using edges. Allowing edges to connect other edges allows for a more explicit representation, such as type links on associations. Both nodes and edges can be accessed using a unique identifier.

Self-connecting edges can be problematic for recursive algorithms, which traverse an edge by going on to the source and target. Therefore, edges can, by construction, only link elements that already exist. Given that the source and target of edges cannot be changed, this effectively prevents loops and guarantees that such algorithms will terminate.

Several types of primitive values are supported. Informally, these types are integers, floating point values, strings, booleans, and action language elements. Action language elements are the primitives of the action language, which is the core of the MvK execution, discussed later (Section 5.10). Supported values are *If*, *While*, *Assign*, *Call*, *Break*, *Continue*, *Return*, *Resolve*, *Access*, *Constant*, *Declare*, *Global*, *Input*, and *Output*. None of the value sets overlap and therefore it is possible to automatically infer the data type.

CRUD interface

An MvS implementation needs to offer the operations defined here, irrespective of its implementation algorithm or data structure. An implementation does not need to use a graph representation, as long as the operations return exactly the same result as if this were the case. We distinguish four types of operations: Create, Read, Update, and Delete (CRUD). For each set of operations, we define the function signature and the required semantics.

Create First are the create instructions, causing the creation of new elements in the graph, thereby extending its size. Each newly created element will be assigned a unique identifier by the MvS, which is returned. It is this identifier which acts as the handle to that element in the MvS. An overview of the operations is shown in Table 4.7. The `create_dictionary` operation is a composite operation for performance.

Operation	Semantics
create_node	Create a new node without value.
create_value	Create a new node with a value.
create_edge	Create a new edge between two elements.
create_dictionary	Create a new dictionary entry.

Table 4.7: Create operations in the MvS.

Operation	Semantics
read_value	Read the value of a node.
read_outgoing	Read outgoing edges.
read_incoming	Read incoming edges.
read_edge	Read source and target of edge.
read_dictionary	Read dictionary entry by key value.
read_dictionary_node	Read dictionary entry by node.
read_dictionary_edge	Read dictionary edge by key value.
read_dictionary_node_edge	Read dictionary edge by node.
read_dictionary_reverse	Read all dictionaries in which it is a value.
read_dictionary_keys	Read all keys of the dictionary.

Table 4.8: Read operations in the MvS.

Read The next set of operations consists of read operations. As there is no information in non-data nodes, there is no read operation defined on nodes, except for their primitive data. An overview of the operations is shown in Table 4.8. Noteworthy are the special `read_dictionary_*` operations, which are specifically tailored to frequent dictionary operations for performance reasons.

Update Even though we implemented a CRUD interface, we do not offer support for any update operations, mostly for correctness and performance.

Edges cannot have their source or target updated, as updating this has the potential of creating loops (e.g., an edge referring to itself). While this was impossible to do when constructing the edge at first, as it is required that its source and target already exist, this can no longer be guaranteed when the edge is updated. If we were to implement such an update function anyhow, we would have to do a lot of sanity checking for whether or not this operation is allowed.

Updating the value of a data node is also not allowed, as this would make read operations non-cacheable. In case values don't change, the values of nodes can be cached in the MvK, meaning we can cut down on requests. Such cache management was also used by WebGME [191, 192].

Delete Finally there are the *delete* operations, of which an overview is given in Table 4.9. After each deletion operation, the resulting graph should be the largest possible subgraph of the original graph, while still being a valid graph. As such, when an element is removed, all outgoing and incoming edges are recursively removed as well.

Operation	Semantics
delete_node	Remove a node.
delete_edge	Remove an edge.

Table 4.9: Delete operations in the MvS.

Summary

We have defined the three core components of the Modelverse architecture: the Modelverse Interface (MvI), Modelverse Kernel (MvK), and Modelverse State (MvS). Each component provides an interface at a varying level of abstraction. The Modelverse Interface (MvI) interfaces with the user, thereby focusing on usability of the Modelverse. Concretely, operations in the graphical interface cause various operations to be executed on the Modelverse Kernel. The Modelverse Kernel (MvK) interfaces with the MvI, thereby focusing on the computational side of the operation. Concretely, a modelling operation is expanded in a list of operations on the graph structure stored by the Modelverse State. The Modelverse State (MvS) interfaces with the MvK, thereby focusing on the storage side of the operation. Concretely, the graph-based operations are processed internally.

Summary

This chapter has introduced the foundation on which the remainder of this thesis is built. First, we introduced the three types of users considered: the modeller, the language engineer, and the Modelverse developer. Second, the definition of MPM was used to distil ten requirements for an MPM tool, which were explained in the context of the power window case study. Finally, a high-level architecture was proposed, considering three components: the Modelverse Interface (MvI), the Modelverse Kernel (MvK), and the Modelverse State (MvS).

Chapter 5

Modelverse Development using MPM

With the specification of the components and their interfaces defined, the development of the components themselves is the next step. While the implementation could be done in many ways, we have motivated the use of the MPM approach. In this approach, we model all relevant aspects of the tool explicitly, at the right level(s) of abstraction, using the most appropriate formalism(s). As such, we describe all aspects of the tool that implements our foundation for MPM: the Modelverse. For each aspect, we describe its purpose, motivation for modelling, present (part of) the model, and provide an evaluation of modelling this part explicitly. We also link back to the requirements to evaluate how this specific component aids in satisfying the previously defined requirements. Note that **Requirement 10 (Portability)** is practically always influenced, as models are naturally less platform-dependent, and is therefore only considered briefly in each section.

Figure 5.1 presents a detailed overview of the Modelverse architecture, similar to the one in the specification. The various components that make up the MvI, MvK, and MvS are mentioned, together with the section where this component is handled. This chapter walks through this overview, presenting each component in turn. Only external tools and the graph database are not modelled explicitly, and as such are not considered further in the remainder of this thesis. Explicitly modelling them would not prove beneficial (e.g., the graph database) or is simply impossible (e.g., the external tool). This still follows the MPM approach, as only the relevant aspects need to be modelled, where there is an obvious advantage of modelling.

Whereas some aspects are relevant to model, the achieved benefits are limited to decreased development time, and as the contribution is minimal. These are still modelled to reduce development time or gain additional minor benefits, but are only briefly mentioned and evaluated in this thesis. These aspects are the Graphical User Interface (Section 5.1), the wrapper (Section 5.2), the network protocols (Section 5.3), the FTG (Section 5.5), the core library (Section 5.4), and the task manager (Section 5.11).

Major advantages of explicit modelling become visible in the other aspects, which are therefore more extensively handled and evaluated in this thesis. The conformance algorithm

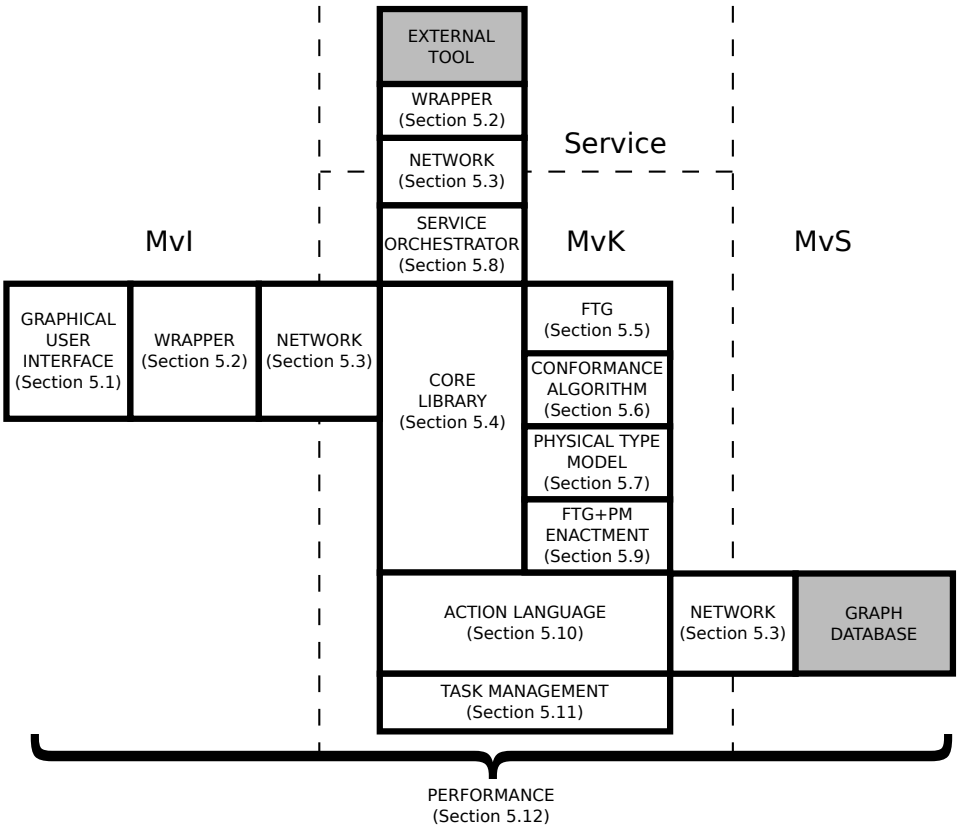


Figure 5.1: Overview of all components of the Modelverse, together with the section in which their model is described.

(Section 5.6) allows for multiple different definitions of conformance, allowing different tool semantics to be emulated. The physical type model (Section 5.7) splits the physical from the linguistic dimension, allowing flexibility at the implementation level. FTG+PM enactment (Section 5.9) maps an FTG+PM to an equivalent model, allowing reuse of SCCD semantics and uniformity. Service orchestration (Section 5.8) models the protocol for external tools, allowing well-defined access to non-modelled components. Action Language (Section 5.10) explicitly models execution semantics, allowing platform-independent code synthesis and documentation. Performance (Section 5.12) models performance of the system, allowing deterministic what-if analysis and increased performance. We handle each aspects in turn, going from the user interface to the low-level details of the Modelverse.

Appropriateness When choosing the most appropriate formalism, we have to consider what it means for a formalism to be "most appropriate". This depends on a variety of criteria. Some of these are objective, such as the verbosity of the language, the expected development time, or the existence of algorithms for that formalism (e.g., reachability graph construction for Petri Nets). Nonetheless, many subjective criteria exist as well, such as the familiarity and intuitiveness of the formalism, or the understandability for some specific

purpose. An important criteria to consider is also pragmatics: how well is the formalism actually supported in actual tools. While some formalisms might boast huge benefits, they might effectively be unusable if there is no stable and efficient tool support.

5.1 Graphical User Interface (GUI)

A Graphical User Interface (GUI) is a visual interface to the underlying tool. In our case, the GUI will be one of the means for users to interact with the Modelverse, in a graphical way. The other ones being the textual, programmatical, or prompt-based approaches. As usual, a GUI is mostly concerned with the usability of the underlying program, and not so much with the actual logic underlying it.

While we only consider the possibility for a GUI here, as this is by far the most complex, different types of interface are indeed possible, and were modelled, for the Modelverse. For example, a frequent discussion in the modelling community is about textual versus graphical modelling [128, 219]. Often this depends on personal preferences and the problem domain being tackled. For example, procedural code is ideally expressed textually, though an architectural drawing of a house is ideally expressed visually. We only consider the GUI, as this is by far the most complex due to the interaction with libraries (e.g., TkInter), timing, and complex events (e.g., mouse events).

Having a GUI is a critical enabler and a determining factor for usability, although it does not contribute to the goals of this thesis. As such, only a proof of concept implementation is briefly considered, although access is provided to all features of the Modelverse.

5.1.1 Motivation

A GUI, or at least some type of user interface, is needed to boost usability. For the Modelverse, a new GUI is created, which is modelled using SCCD [298], thereby following the MPM approach. As the primary advantages are related to development time and portability, we only consider this aspect briefly.

Why a new GUI?

A plethora of graphical interfaces already exist for (meta-)modelling tools, and thus it might seem wasteful not to reuse any of them. Our GUI is only to be considered as a prototype implementation, offering all features that the Modelverse has to offer. Other GUIs are often tightly interwoven with their accompanying back-end, making them difficult to refactor and couple to the Modelverse. For example, the AToMPM client (JavaScript based) could not easily be reused, as the operations and model exchange protocol is convoluted and non-trivial. Similarly, existing GUIs are tailored to the (meta-)modelling tool they accompany, and therefore have no (preliminary) support for other features. For example, process enactment requires spawning a new (restricted) editor with a specific model pre-loaded, receiving a notification when the editor has finished. Such functionality is not yet supported by any of the tools considered. While these interfaces could certainly be modified to include this functionality, this will likely take more time than creating a completely new prototype. Advantages of the new GUI are that it is then completely tailored to the full featureset, and the new features are not just hacked in. We must then accept, however, that the GUI will not be of the same level of maturity as existing GUIs.

Why model the GUI?

GUI development is usually associated with lots of accidental complexity, of which we consider two aspects: complex event handling (e.g., mouse events, key events, internal UI updates, window manager updates, timings, and so on) and varying widget library between different platforms. To this end, modelling the GUI proved simpler than coding this with triggers and callbacks, as is frequently the case.

Why model the GUI with SCCD?

SCCD was selected as the most appropriate formalism for this problem, as it provides native constructs to the previously mentioned sources of complexity. Complex event handling is managed natively by the discrete event base nature of SCCD. Used widgets can also be implemented as a dedicated SCCD class, making them applicable in many applications (they can be fully tailored) and independent of the provided widget library. Finally, SCCD has mature code generation capabilities for a variety of platforms, including bindings with the TkInter's event loop [298]. Note that the GUI does more than only present a set of widgets, and a widget domain-specific language is therefore not appropriate. Indeed, many widgets influence one another (e.g., pressing the "open" button creates a new toolbar of widgets), and the behaviour would thus be hard to describe in an isolated widget.

5.1.2 Model

Figure 5.2 shows an excerpt of the GUI SCCD model, focussing on the *CanvasElement* class. The *CanvasElement* class provides the visualization of a model element (e.g., a circle for a Petri Net place). This element can be dragged around, removed, attributes can be modified, links can be created, and so on.

The Class Diagrams part is partially shown, showing how this class relates to the initial class *MainWindow*. From the *MainWindow*, several associations exist to other classes, such as *Toolbar*, *ProgressBar*, and *Canvas*. Naturally, the *CanvasElement* is associated to a *Canvas*. The *CanvasElement*, in its turn, has an association to the *PromptWindow*, as it must be able to prompt the user for information, for example when modifying the attributes of an element. Many other classes are not shown for clarity, though many inherit from *Window* or *Toolbar*. Our custom-built widgets are also modelled explicitly using classes, such as *Entry*, *Text*, and so on.

The Statechart is only shown for the *CanvasElement*, in which actions and conditions are abstracted for readability. We now briefly explain the behaviour of the *CanvasElement*. Upon initialization, we first check the type of rendering: either direct (as most other tools) or through perceptualization (see Section 6.3). If it is direct, we immediately go to *main*, where we wait for input. Otherwise we go to *update_MV* to update the *x* and *y* attribute (position) in the Modelverse. In the *main* state, there are several options, as shown in the Statechart. Either an internal event can come in to draw the element, or different clicks can occur. On middle click with the control key pressed, we modify the defined attributes, which happens outside of the *CanvasElement*. On middle click without the control key pressed, we modify the actual attributes, for which we go to the *update_attrs* state. Here, we query the attributes and open a prompt which allows users to modify these values. This prompt is handled by a different class, and we only wait for an event indicating that the prompt has been closed. When closed, changes are assigned in the Modelverse and

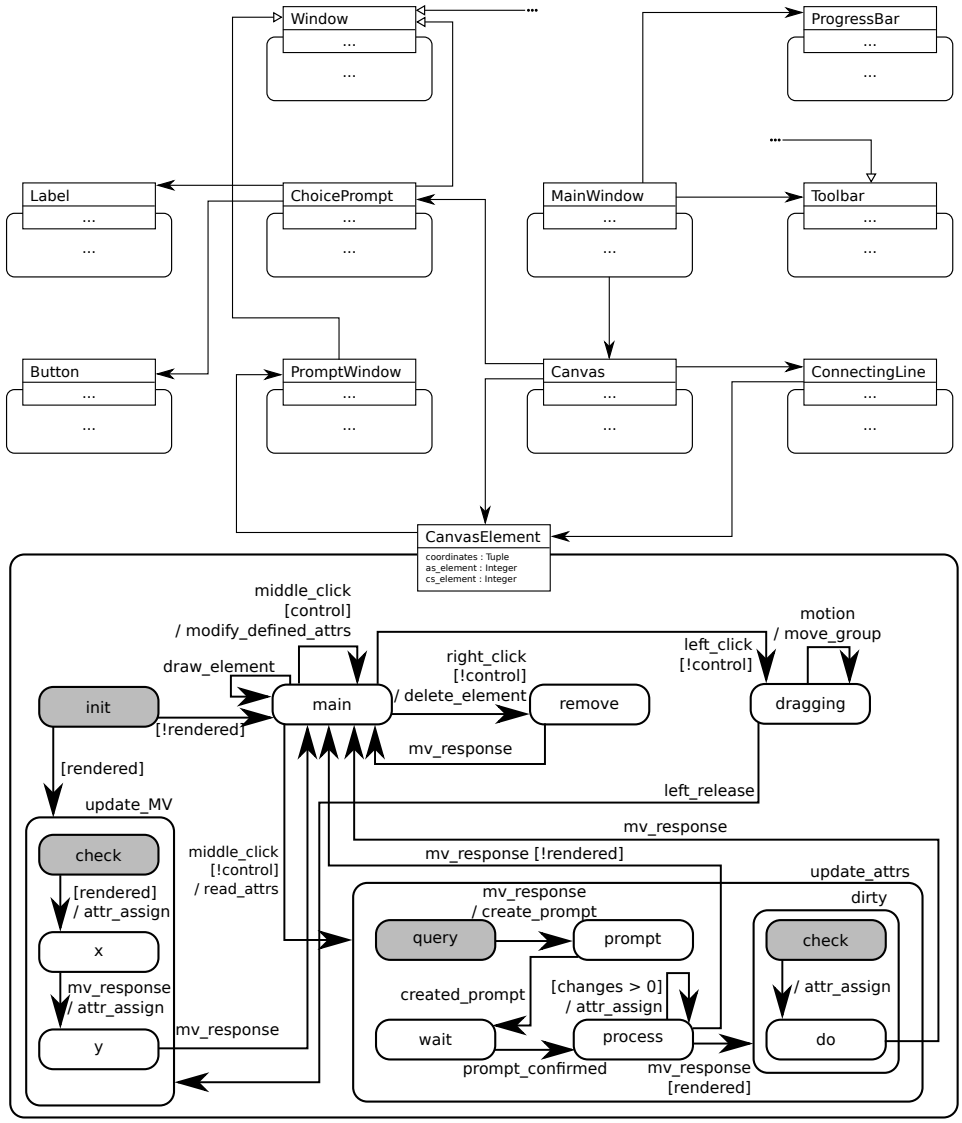


Figure 5.2: SCCD model of the GUI (excerpt).

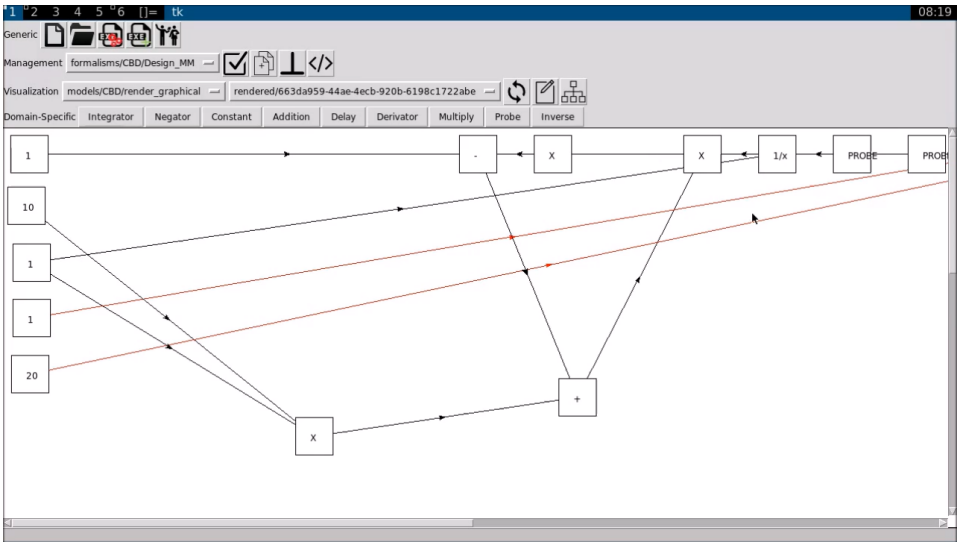


Figure 5.3: Screenshot of the GUI with a loaded CBD model.

(optionally) this element is marked as *dirty*, meaning that it should update its position in the Modelverse. Back in the *main* state, a right click on an element causes it to be deleted. A left click starts dragging the element, where it responds to motion events by sending out a *move_group* event to the complete group it is contained in. After the left mouse button is released, the *x* and *y* attributes are updated in the Modelverse again.

A screenshot of the running GUI is shown in Figure 5.3, where a simple CBD model is visualized and can be manipulated. This screenshot also shows the different toolbars: the generic toolbar (new model, open model, enact process, ...), the model management toolbar (alter metamodel, check conformance, ...), the visualization toolbar (different visualizer, different visualized model, refresh visualization, ...), and the domain-specific toolbar (containing domain-specific concepts to instantiate).

5.1.3 Evaluation

We now briefly evaluate the SCCD model created for the GUI with respect to our original motivation. No detailed evaluation is given, as there is no scientific contribution.

By not relying on an existing GUI, we more prominently display the new functionality that is offered by the Modelverse. For example, it was natural to give a prominent place to the multiple notions of conformance (Section 5.6), conformance bottom (Section 5.7), and process enactment (Section 5.9). The newly created GUI is however not familiar to modellers and it is centered around showcasing new features, instead of usability.

By modelling the GUI using SCCD, event processing did indeed become easier. As shown in the *CanvasElement*, we make extensive use of GUI events as triggers for transitions. Documentation-wise, this model clearly indicates what happens when an element is left clicked (*left click* event), dragged around (*motion* event), and then released (*left release*

event). This is in contrast to three callback handlers, each of which registered to an object and having to do mode checks themselves.

Each widget has its own SCCD implementation, instead of relying on the default widgets of TkInter. While these widgets are sometimes still used, their manipulation is completely event-based. For example, when clicking a button, no callback handler is invoked, but an event is raised to the widget's parent. This makes it more flexible to switch between implementations, for example to port the same interface to PyGTK+, which has a different set of widgets, with a different interface and different behaviour. Similarly, porting the application between programming languages should also be simpler, as both the modal and structural code has been shifted to SCCD. Only non-modal code remains, although this is written in Python due to limitations of the SCCD compiler: there is no neutral action language which allows translation to different platforms. While the porting process is not completely automated, the effort is still reduced, as only a subset of the code has to be ported.

5.1.4 Link to Requirements

Explicitly modelling a minimal GUI has an influence on **Requirement 6 (Multi-Interface)** and **Requirement 10 (Portability)**.

Requirement 6 (Multi-Interface) is influenced because the SCCD model serves as a starting point for other interfaces, given that it implements all features of the Modelverse. As mentioned before, several other interfaces have been created based on this same interface. While all these interfaces are admittedly simple and unpolished, they show that such an interface can easily be constructed. Additionally, it shows that multiple such interfaces can jointly operate on the same Modelverse.

Requirement 10 (Portability) is influenced by the (partial) platform-independence of the underlying SCCD model. For the modal and structural code, there is no dependency on the implementation language, such that code for different platforms can automatically be synthesized.

Summary

A (Graphical) User Interface (GUI) is an important factor in the usability of (meta-)modelling tools. For the Modelverse, we explicitly modelled a proof of concept implementation of a GUI using SCCD, making nearly all functionality of the Modelverse available. SCCD was judged to be an appropriate formalism due to its native constructs for concurrency, event processing, and timing, as well as code synthesis capabilities. Parts of the SCCD model were presented, giving an idea of the documentation value of this model: the behaviour is intuitively clear, in contrast to the many convoluted callbacks that would otherwise be necessary. The modal behaviour of the GUI code could be automatically synthesized, thereby decreasing development time and increasing portability.

5.2 Wrapper

As there are multiple (types of) interfaces to the Modelverse, we cannot expect each interface to reimplement the Modelverse protocol from scratch. The Modelverse be-

ing a service, it has its own communication protocol, specifying supported operations, their signature, their return value, and possible exceptions. A seemingly atomic operation in the client is often translated to several requests to the server. For example, the API function `model_add(metamodel_name, model_name, HUTN_code)` results in several Modelverse requests: `model_add`, `metamodel_name`, `model_name`, and `HUTN_code`. Additionally, some client-side checks are also performed before the call is made, such as whether or not the HUTN_code represents a string. Through these checks, we can at least guarantee the syntactical validity of the client's request.

Having a reusable implementation of this protocol makes interfacing with the Modelverse much simpler, as it can be invoked as if it were a local operation. It is thus useful to define this independent of the interfaces, offering a reusable, high-level interface to the Modelverse to all user interfaces. Other interfaces, such as the Python API, easily reuse the exact same protocol.

5.2.1 Motivation

A reusable Modelverse protocol wrapper is required to speed up development across multiple Modelverse Interfaces. For the Modelverse, we explicitly modelled the protocol as an SCCD model, thereby following the MPM approach. As the primary advantages are related to development time and portability, we only consider this aspect briefly.

Why model the wrapper?

Similar to the GUI, protocol development is not ideal with existing programming languages. Indeed, protocols rely on different modes, timeouts, and asynchronous events, making them non-trivial to implement and verbose. Additionally, coding the protocol makes it non-portable between different implementation platforms.

Why model the wrapper with SCCD?

Similar to the GUI, the reliance of protocols on states, timeouts, and asynchronous events makes the match with SCCD easy to make. Indeed, finite state machine representations of a protocol are common as documentation in standard documents (e.g., Figure 6 in RFC793[1]). The code synthesis feature of SCCD again enables portability between different interfaces, even if they are not written in the same programming language. While technically Statecharts would suffice as well, SCCD was used to make the code more interoperable with other SCCD code, in particular the GUI. (Timed) Finite State Machines do not suffice, as they have no notion of concurrency.

5.2.2 Model

Figure 5.4 presents a significantly abstracted version of the client wrapper statechart. The main motivation for why a Statechart was required is still visible. The full model can be found in the Modelverse Interface source code.

The leftmost part of the Statechart indicates the different modes the client is in. The client can be *initializing* (i.e., setting up sockets and such), *connected* (i.e., ready for login), in

[1] `https://tools.ietf.org/html/rfc793`

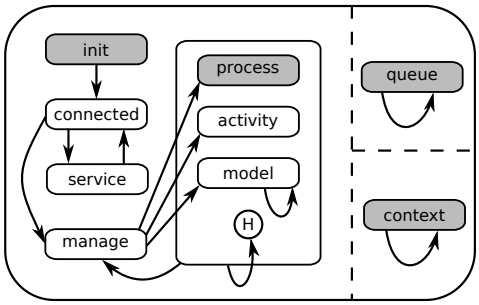


Figure 5.4: Abstract version of the client wrapper Statechart.

service mode (i.e., raw communication with an activity in the Modelverse), in *management* mode (i.e., at the megamodel level), or can be *modifying* a model. In each of these different modes, different operations can be performed by the client (e.g., login only when in *connected*). There are three ways in which the model can be modified: as part of an enacted *process* (where the interface is limited), through an *activity* (where raw communication happens with the client), or through the usual *modelling* operations (most common). In the modelling state, mostly the same set of operations is supported, after which we return to the history state, as we have to stay in the same state. Some operations are only available when modifying the model directly, and some operations exit the currently opened model. The rightmost part of the Statechart indicates two orthogonal components: the queueing of inputs (rendering behaviour asynchronous to the user), and the processing of context events (which allow the client to update its state with that of the Modelverse).

5.2.3 Evaluation

We now briefly evaluate the SCCD model created for the client wrapper with respect to our original motivation. No detailed evaluation is given, as there is no scientific contribution.

Even though Figure 5.4 is an abstracted view on the actual model, which is much larger, it still highlights the advantages of using Statecharts. First, the interface operates asynchronously from the client, therefore requiring orthogonal states. An asynchronous interface is required, as otherwise the interface (e.g., a GUI) would hang while the operation is processed. Creating an asynchronous interface is much harder without Statecharts, as it would require threads. Threads conflict with the use of TkInter's eventloop, meaning that the notion of concurrency would have to be updated depending on which context the protocol is used in.

Second, the different modes of the interface are mapped to the states of Statecharts. In each states, different transitions are defined, reacting to a specific set of input events. This is easier to conceptualize than having a single large if/then/else block. Additionally, entry and exit actions can be defined on each of these states, making sure that some action certainly occurs when switching modes (e.g., closing the model when going back to *manage*).

Third, several native constructs of Statecharts significantly help, such as the history state and transition timeout. For the history state, the current state has to be memorized when doing operations, for which the history state proves useful. For the timeouts, this can be

easily used to handle an unresponsive server, for example because of a lost connection to the Modelverse.

5.2.4 Link to Requirements

Explicitly modelling the wrapper to the Modelverse protocol has an influence on **Requirement 6 (Multi-Interface)** and **Requirement 10 (Portability)**.

Requirement 6 (Multi-Interface) is influenced, as the protocol can be reused across different interfaces. This decreases the time required to create a new interface, and makes sure that all interfaces use the same protocol.

Requirement 10 (Portability) is influenced, as the protocol was modelled and therefore platform-independent. For example, we have shown the advantage in combination with different concurrency implementations (TkInter versus threads). Additionally, SCCDs code synthesis capabilities can generate code for different programming languages as well, with the exact same behaviour.

Summary

As the Modelverse has a client-server architecture, its operations and their signature follow a specific protocol that has to be implemented in each interface. We explicitly modelled this protocol using SCCD, which has native constructs for the concepts required: timeouts, history states, concurrency, and states. This model can be used for documentation purposes, but also for code generation for different platforms, handling different concurrency implementations (e.g., TkInter versus threading) and different programming languages.

5.3 Network Protocols

The use of a client-server architecture naturally requires the presence of a network inbetween the client and server. While the Modelverse protocol was defined in the previous section, these requests still need to get to the Modelverse over the network intact and in the correct order. This is the job of the network protocol: requests are serialized and transferred following some protocol, for example XML/HTTPRequests or WebSockets. At the server-side, the data is received and deserialized again.

For maximum portability across languages and the least problems with existing firewalls, we have chosen to use XML/HTTPRequests. This choice is particurly relevant between the MvI and MvK, as this will likely be used over high latency networks with restrictions. While a network protocol also has to be chosen between the MvK and MvS, this has completely different requirements, as we have full control over the platform, the firewalls, and latency is likely to be low. For simplicity, XML/HTTPRequests are used throughout. These types of requests are generally better supported in current libraries and tools, and have a higher change of being allowed even with strictly configured firewalls. Additionally, generic XML/HTTPRequest tools, such as web browsers, can trivially communicate with the Modelverse. Nonetheless, this does mean that more advanced functionality (e.g., server-initiated communication) has to be implemented on top of this minimal protocol.

5.3.1 Motivation

We have chosen to model the network communication protocol from scratch using SCCD. As the primary advantages are related to development time and portability, we only consider this aspect briefly.

Why a new network protocol library?

Network communication, and in particular using XML/HTTPRequests, is frequently used today, and various libraries exist for it. However, all these libraries have a different focus, different interface, and often a different set of features. For example, the implementation in the Python standard library is limited to synchronous requests and non-threaded execution, which has a negative impact on performance and usability. Indeed, we do not want that the GUI is blocked for each networked request. Other libraries support asynchronous communication, but then these libraries only exist for a single language and cannot be ported (e.g., to Java). Combined with widely varying interfaces, it proves difficult for the wrapper to interface with the library, as each platform requires its own implementation and workarounds. Given the simplicity of the XML/HTTPRequests protocol, this protocol was implemented from scratch to provide a reusable implementation across all platforms.

Why model the network protocol?

Similar to the wrapper, the network protocol implements a protocol, and therefore we expect the same benefits we saw for the Modelverse wrapper: easier to implement and increased portability.

Why model the network protocol with SCCD?

As with the Modelverse wrapper, SCCD is an appropriate formalism due to its native support for states, events, and timeouts. The dynamic structure features of SCCD are also useful, as the number of socket instances varies throughout time. The code generation capabilities of SCCD can again be used to generate code for various platforms.

5.3.2 Model

We now present the model of the network protocol used for the Modelverse. We only present an abstract view for readability, ignoring actions and abstracting conditions, while the states and their transitions are fully shown. Two components are considered: the server and client side. In both cases, the full model can be accessed from within the Modelverse source code.

Contrary to the standardized HTTP specification, we allow for keep-alive connections by default: when data is received, the same socket can again receive input or send back output. Originally, HTTP specifies the use of a new socket for each request, thereby incurring a high overhead. Newer versions of the HTTP protocol (optionally) support keepalive as well, though this has not yet been implemented in all HTTP libraries yet. In both the client and server, we remain backwards compatible, meaning that if the other party closes the socket, we also close the socket and open a new one when necessary.

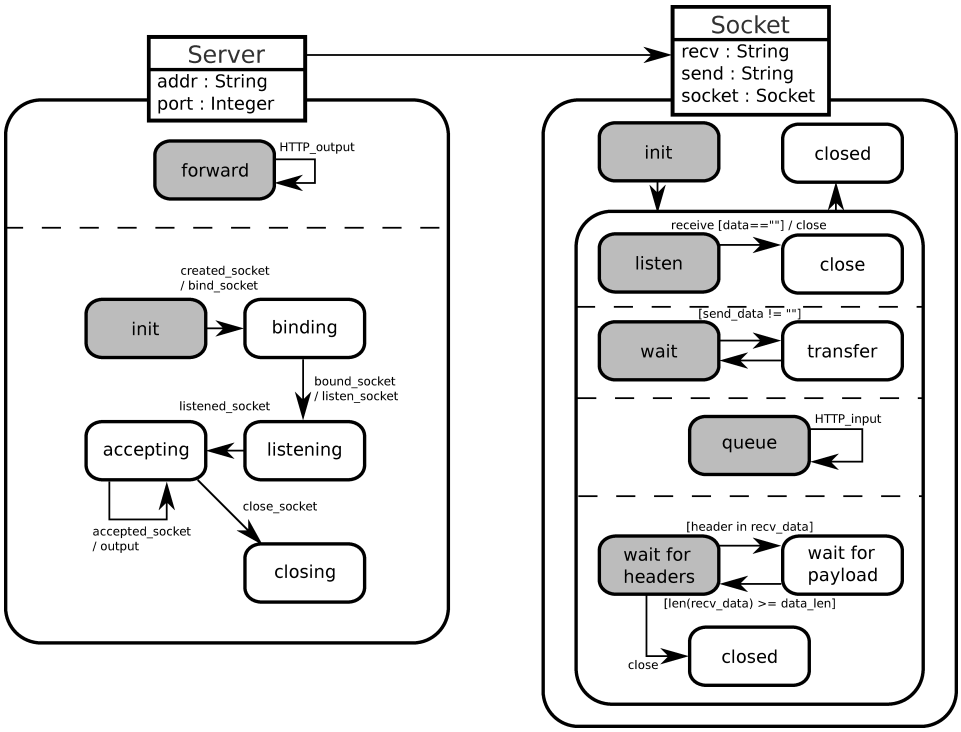


Figure 5.5: Network protocol SCCD model at server side (abstracted).

A minimal wrapper was created that wraps around the Python socket interface. When socket events are generated (e.g., *bind_socket*), the respective Python socket function is called asynchronously. Asynchronicity is important, as actions in Statecharts are assumed to take zero time. This socket library has to be reimplemented on the different platforms that we consider.

Server-side

Figure 5.5 presents the SCCD model used to model the network protocol at the server side. There are two classes in this SCCD model: the Server and the Socket. The Server is the main component that provides the interface to the user of the library. Upon startup, it creates a server socket on which the system listens for new connections. When a new connection is opened, a Socket object is spawned, which then starts communication. In the Socket object, received data is appended to a queue until the header is completely received. Using information from the header, the remainder of the input data is used to decode the actual payload. After the payload is received, we return back to the listening state, again waiting for a header. When data has to be sent back by the server, this is first put into a queue, which is later processed by an orthogonal component.

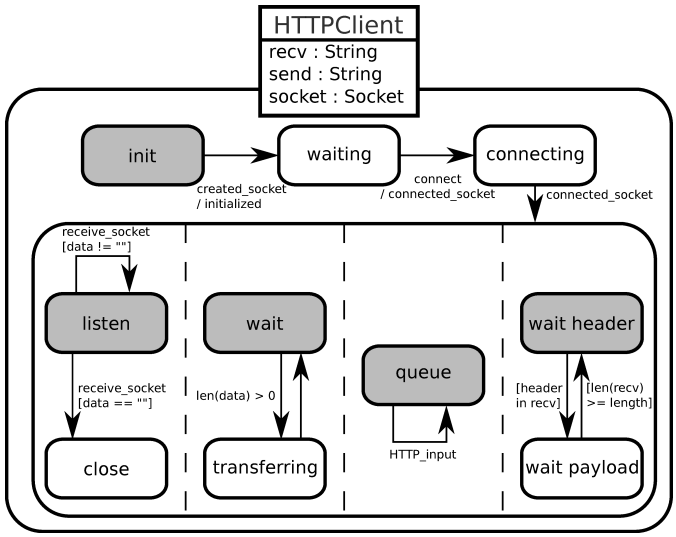


Figure 5.6: Network protocol SCCD model at client side (abstracted).

Client-side

Figure 5.6 presents an abstracted version of the SCCD model used to model the network protocol at the client side. There is only one class in this SCCD model: the HTTPClient. The client is started and immediately creates a socket and prepares to set up the connection. As soon as everything is ready, an event is issued to the instantiator to notify that the client is ready for input. Subsequently the "connect" event can be sent to make the connection to the specified address. After the connection is made, we enter the orthogonal component. Here, there are four concurrent operations: 1) listen to any incoming message on the socket, which is appended to the input buffer; 2) check if there is sufficient data on the output buffer, and if so, send out the first message; 3) queue any incoming send events from the instantiator of the object; and 4) check if there is sufficient data in the input buffer, and if so, parse the data and send out the reply to the instantiator.

5.3.3 Evaluation

We now briefly evaluate the SCCD model created for the network protocols with respect to our original motivation. No detailed evaluation is given, as there is no scientific contribution.

For both the server and client side, a socket wrapper is used, which provides platform-independent access to low-level socket operations. This is a first step towards cross-platform portability. Sadly, as mentioned before, SCCD does not support a neutral action language at the moment, and therefore Python was used in the body of the functions. Nonetheless, all modal and structural parts of the system are modelled explicitly and therefore portable. Through the use of this SCCD model, not only the GUI, but all SCCD-based applications have asynchronous access through an SCCD-native HTTP library.

Because we have control over both the client and server, we were able to augment the protocol with a keepalive feature: sockets are not closed but reused across multiple requests

and replies. This significantly reduced the overhead of socket connection and binding, while also minimizing the number of sockets being used. When either the client or server do not implement the keepalive feature, we fall back to the original HTTP specification where the connection is closed after the request and subsequent reply.

Thanks to SCCD, the class diagram is used to manage the multiple active sockets, each with their own state.

5.3.4 Link to Requirements

Explicitly modelling the network protocols used by the various components has an influence on **Requirement 4 (Multi-User)**, **Requirement 5 (Multi-Service)**, **Requirement 6 (Multi-Interface)**, and **Requirement 10 (Portability)**.

Requirement 4 (Multi-User) is influenced, as the network protocol is also responsible for the queueing discipline of requests and replies. To ensure that all users get a fair chance of getting a message through (i.e., resource sharing of the network), the network protocol implementation must take this into account. By reusing the same socket all the time for a single user, we limit the amount of sockets opened on the server.

Requirement 5 (Multi-Service) is influenced, as these various services all make use of the same network protocol. As such, it was important to choose a network protocol which all services can support. For that purpose, XML/HTTPRequests were ideally suited.

Requirement 6 (Multi-Interface) is influenced, as the protocol can easily be reused across multiple interfaces. This is similar to the previous requirement, since we also want to maximize the number of interfaces that can connect.

Requirement 10 (Portability) is influenced, as the protocol can be reused across multiple interfaces and on different platforms, independent of library support for these platforms.

Summary

Due to the distributed nature of the Modelverse, network protocols are required. We opted for XML/HTTPRequests due to their wide support: there are tons of libraries and tools supporting it and this protocol is generally allowed through firewalls. We explicitly modelled this protocol in SCCD to maximize the uniformity across multiple interfaces and to guarantee that we can make use of the keepalive optimization. The model gave us documentation on the behaviour of the protocol as well as an automated way to synthesize code out of it.

5.4 Core Library

To enable the model management capabilities of the Modelverse, a wide set of such operations have to be implemented. This is done through the Modelverse model management library, offering the necessary operations: basic modelling operations (e.g., create instance, assign attribute), meta-modelling operations (e.g., define new type, define attribute), and model management operations (e.g., model merge, check conformance). Our library is flexible and can be augmented throughout the lifetime of the Modelverse.

These functions are offered through a master algorithm, which implements the Modelverse protocol with which the client's Modelverse wrapper communicates. It is this function that is responsible for invoking the correct operations based on the incoming requests. This model is thereby also responsible for which internal functions are offered. The model can easily be altered to support new functions or remove existing functions on-the-fly.

5.4.1 Motivation

As was motivated, a core library describing the Modelverse protocol and its supported model management operations is required. We modelled a new model management library using the procedural Action Language of the Modelverse (Section 5.10). As the primary advantages are related to documentation, development time and portability, we only consider this aspect briefly.

Why a new (meta-)modelling library?

Several existing libraries already provide support for modelling, meta-modelling, and model management operations. Examples include EML [165] for model merging and ECL [163] for model comparison. And while these languages and libraries are well established and mature, they are not reusable in the Modelverse project due to their lack of portability and flexibility.

For portability, most existing languages, such as EML and EOL, ultimately depend on Java. For example, it is possible to integrate Java code in EOL, and the reference interpreter is also implemented in Java. This makes it difficult to port such libraries between platforms, as they require a (specific version of the) Java interpreter.

For flexibility, these languages are based on a single approach, hardcoded in the language. For example, many approaches to model merging [54] exist, but EML only implements some of them. While indeed new languages can be conceived that handle more cases, this requires potentially breaking changes to the language, thereby requiring changes to the interpreter. Additionally, it is impossible to delve deeper, or "step", into these languages, as they are seen as atomic. Ideally, we would like to see under the hood what happens when the operation is performed.

Why model the core library?

The primary reason to model the core library, instead of coding it with an existing language, is to allow for increased platform independence. We additionally notice increased uniformity: if the core library is modelled itself, it can be used as any other model. This not only offers introspection by querying the model, but all operations from the core library can also be used as activities, for example in the FTG+PM.

Interestingly, if there is a live model of the core library in the Modelverse, this means that it can be updated dynamically. Indeed, the semantics of all operations could then be updated while the application was running, allowing for on-the-fly bugfixes and patches. Additionally, in theory multiple implementations could live side by side, as models, allowing for an easy way to switch between different semantics for a single operation.

Why model the core library with Action Language?

The question remains why our custom Action Language was used to model this library. In the limit, code can be seen as a modelling language as well, as an abstract syntax, concrete syntax, and semantics can be constructed. Our custom Action Language has several advantages, most of which are discussed later on. For this specific problem, it was important that the Action Language resembles procedural code (arguably the most appropriate for model operations), is minimalistic though expressive (meaning that almost all operations are themselves library functions written in the Action Language itself), and is platform independent (meaning that there are no escape hatches to the implementation language. The minimalism is required to allow very deep inspection of the semantics of the library, resulting in only a few "primitive" operations that cannot be inspected further. An example of this is shown in the model. The platform independence is required to make the library applicable in all possible situations, independent of the implementation platform being used. Indeed, if there was an escape hatch to a general purpose programming language, every implementation of the Modelverse would have to intimately rely on this programming language.

5.4.2 Model

The core library merely consists of several Action Code files, containing about 7,000 lines of Action code in total. This is all it takes to form the core of the Modelverse's behaviour. Of these, approximately 3,000 lines are related to the Modelverse protocol with all associated error handling and access control checking. While this does not include any advanced model management operations, these can be modelled using the provided codebase.

For brevity, only one example operation is shown here, together with its interface: the creation of a new element in a model (*instantiate_node*). The Modelverse core library starts up the main loop, from where all execution originates, shown in Listing 5.1. Input from the user is received and stored in a variable (line 5). The value in this variable determines the command to execute, which in our case finds `model_modify` (line 10). As a result, the function `cmd_model_modify` is invoked with two parameters, which both block for user input: the model in which we perform the modification and the metamodel to use. When these two parameters are passed by the user, as individual inputs, the function is invoked.

```
1  output("Welcome to the Model Management Interface v2.0!")
2  output("Use the 'help' command for a list of possible commands")
3
4  while (True):
5      cmd = input()
6      if (cmd == "help"):
7          output(cmd_help())
8      ...
9      elif (cmd == "model_modify"):
10         output(cmd_model_modify(single_input("Model name?"), single_input("Metamodel
       name?")))
11     ...
```

Listing 5.1: Core library root interface function (excerpt).

Upon invocation of the `cmd_model_modify` function, shown in Listing 5.2, the function starts executing. Checks are now performed to determine whether the passed parameters are sane. First, we check whether the model name exists (line 7) and if current user has read

permission (line 8). Second, the metamodel is subjected to the same tests (line 11). When these tests pass, the full model is read out (line 13) and passed on to the modify function (line 16). During this call, it is determined whether the model is opened in read-only mode, or if writes are allowed. When finally the function returns, the altered model is written out for real (line 19). This prevents concurrent conflicting changes.

```
1  String function cmd_model_modify(model_name : String, metamodel_name : String):
2      // Model modify operation, which uses the mini_modify.alc operations, though with
         extensions for access control
3      String model_id
4
5      model_id = get_entry_id(model_name)
6
7      if (model_id != ""):
8          if (allow_read(current_user_id, model_id)):
9              type_id = get_entry_id(metamodel_name)
10             if (type_id != ""):
11                 if (allow_read(current_user_id, type_id)):
12                     Element new_model
13                     new_model = get_full_model(model_id, get_entry_id(metamodel_name)
       )
14                     if (element_eq(new_model, read_root())):
15                         return "No conformance relation can be found between these
       models"!
16                     modify(new_model, allow_write(current_user_id, model_id))
17                     if (allow_write(current_user_id, model_id)):
18                         // Overwrite the modified model
19                         model_overwrite(new_model, model_id, get_entry_id(
       metamodel_name))
20                     return "Success"!
21                 else:
22                     return string_join("Permission denied to model: ", full_name(
       type_id))!
23             else:
24                 return "Metamodel not found: " + metamodel_name!
25         else:
26             return "Permission denied to model: " + model_name!
27     else:
28         return "Model not found: " + model_name!
```

Listing 5.2: Core library modify permission check function (excerpt).

We now delve into the `modify` function, shown in Listing 5.3. The design is similar as the previous interface function: a big *If* construct with all supported operations. In our case, the *instantiate_node* function is selected in line 12. The type and name of the operation to perform are also read out and passed on to the interface function.

```
1  Boolean function modify(model : Element, write : Boolean):
2      String cmd
3
4      output("Model loaded, ready for commands!")
5
6      while (True):
7          cmd = input()
8          if (cmd == "help"):
9              output(cmd_help_m(write))
10         ...
11         elif (cmd == "instantiate_node"):
12             output(cmd_instantiate_node(write, model, single_input("Type?"),
       single_input("Name?")))
13         ...
14
15     return True!
```

Listing 5.3: Modification sub-interface (excerpt).

The function `cmd_instantiate_node` again performs some checks whether or not the operation is valid. In this case, we check whether we are allowed to modify the model (line 2), whether the type exists (line 3), whether the element already exists (line 4), and whether the type we are instantiating is not an edge (line 7). If all checks pass, the actual library function is called with the provided parameters (line 10).

```
1  String function cmd_instantiate_node(write : Boolean, model : Element, mm_type_name :
       String, element_name : String):
2      if (write):
3          if (dict_in(model["metamodel"]["model"], mm_type_name)):
4              if (dict_in(model["model"], element_name)):
5                  return "Element exists: " + element_name!
6              else:
7                  if (is_edge(model["metamodel"]["model"][mm_type_name])):
8                      return "Element is not a node but an edge: " + mm_type_name!
9
10                 element_name = instantiate_node(model, mm_type_name, element_name)
11                 return "Success: " + element_name!
12         else:
13             return "Element not found: " + mm_type_name!
14     else:
15         return "Permission denied to write"!
```

Listing 5.4: Modification pre-check (excerpt).

The library function `instantiate_node` is also modelled explicitly in the Modelverse, and the source code is present shown in Listing 5.5. Here, most checks are guaranteed to be done, so no additional checks take place. The actual instantiation is done in three steps: 1) a new unique model element ID is generated, taking into account the desired ID (line 6), 2) the newly created node is added in the model dictionary with the provided key (line 7), and 3) the node is (re)typed to the desired type (line 8). At the end, the actually assigned ID is returned, which might not be identical to the requested ID if that was already taken. The individual functions called here, are also modelled explicitly themselves, though this would lead us too far. All functions mentioned here eventually end up in *primitive* operations (such as `dict_add`), which don't have an explicit model, but which are trivially mapped to MvS operations (such as *create_dict*). About 60 such primitive operations exist, all with an intuitive mapping to MvS operations.

```
1  String function instantiate_node(model : Element, type_name : String, instance_name :
       String):
2      String actual_name
3      Element value
4
5      value = create_node()
6      actual_name = instantiated_name(value, instance_name)
7      dict_add(model["model"], actual_name, value)
8      retype(model, actual_name, type_name)
9      return actual_name!
```

Listing 5.5: Instantiate node model operation (excerpt).

5.4.3 Evaluation

We now briefly evaluate the Action Code model created for the core library with respect to our original motivation. No detailed evaluation is given for this topic, as this does not present a significant contribution apart from development time and portability.

By modelling the core library explicitly, we can query the core library without additional tools. Indeed, the core library is itself a model, and can thus be opened and manipulated like

any other model. The library can even be rewritten dynamically, making it possible to patch the Modelverse operations while it is running. Only primitive operations cannot be altered, though these are trivial and tightly interwoven with the MvS operations. Additionally, most operations can be used directly as part of activities.

The explicitly modelled core library also makes the Modelverse highly portable: everything is based on a minimal Action Language, for which an interpreter is trivial to create (Section 5.10). As such, the core library is independent of any platform and can easily be ported to a different platform. This is reflected in the functions: everything is explicitly modelled up to the level of the raw graph manipulation operations. As such, the internals of the tool can be inspected and potentially manipulated, thereby aiding in, for example, debugging of the core library.

5.4.4 Link to Requirements

Explicitly modelling the core library with model operations has an influence on **Requirement 1 (Language engineering)**, **Requirement 2 (Activities)**, **Requirement 3 (Process modelling)**, **Requirement 4 (Multi-user)**, **Requirement 6 (Multi-interface)**, and **Requirement 10 (Portability)**.

Requirement 1 (Language engineering) is influenced, as all language engineering operations are implemented using this core library. When different language engineering features are required (e.g., multi-level modelling), this can easily be done by altering the core library directly, without restarting the Modelverse at all.

Requirement 2 (Activities) is influenced, as all model management operations, for which an implementation exists in the core library, are now explicitly modelled. Using this explicit model, it becomes possible to use them as activities as well, or use them directly as part of other activities. Furthermore, it is this core library that is responsible for executing the activities.

Requirement 3 (Process modelling) is influenced, as all enactment of the process is based on operations in this core library. Indeed, the function `process_enact` is part of this core library.

Requirement 4 (Multi-user) is influenced, as resource allocation for these users can only happen at the level of action language. Should an operation not be modelled like this, it has to execute as a single atomic instruction, for the Modelverse has no way of pausing its execution (i.e., pre-emption).

Requirement 6 (Multi-interface) is influenced, as an interface optionally has the chance to extend the existing Modelverse interface by defining new operations in the core library. As this is done dynamically, this can be done without negative effects for other interfaces.

Requirement 10 (Portability) is influenced, as an explicitly modelled core library makes the tool independent of existing model management libraries. While it is a cost to implement this library, all operations are fully portable, without relying on existing implementations.

Summary

The meta-modelling library encompasses all supported modelling, meta-modelling, and model management operations of the Modelverse, as well as its protocol. While several such libraries and languages exist, we have explicitly modelled a new library using the Action Language of the Modelverse, creating complete platform independence and allowing dynamic updates at runtime. Our custom Action Language was used as a formalism due to its platform independence and minimality, making it ideally suited to delve deeper into the internals of the operations. More importantly, explicitly modelling these operations allowed them to be considered as ordinary models, enabling model queries and their use as activity.

5.5 Formalism Transformation Graph

The Formalism Transformation Graph (FTG) describes all formalisms present in the Modelverse and the activities between them. This offers an overview of all the models, formalisms, and activities that can be (re)used by modellers and language engineers. Activities provide a means of going from one formalism to another, while retaining certain properties. For example, a domain-specific language with an activity mapping it to Petri Nets, can reuse the analysis of Petri Nets for its own.

In the Modelverse, this FTG is additionally used to organize and structure all present models. Models in the FTG are annotated with meta-data, storing information about permissions, traceability, properties, and so on. Similarly, activities can store their input and output signature, describing which models they operate on.

5.5.1 Motivation

The FTG provides structure to the plethora of models that live in the Modelverse. Instead of coding this and keeping the structure hidden in the code, we modelled this explicitly as a megamodel. As the primary advantages are related to decreased development time due to reuse, we only consider this aspect briefly.

Why model the FTG?

By turning the FTG into a model, it can be manipulated like any other model, thereby circumventing the need for new operations. Indeed, deleting a model is now just removing an element in the FTG model, reusing much of the existing code of the core library. Querying the available models and their transformations is similarly identical to existing model management operations. Furthermore, the FTG is a data structure as well, making it unnecessary to store model references in a coded data structure. When modelled, even multiple FTGs are possible, for example one for every user, based on the access permissions of that specific user.

Why model the FTG as a megamodel?

A megamodel is the natural choice for the FTG, as indeed the FTG is a model containing (or referencing) other models. By storing references to each model in another model, the

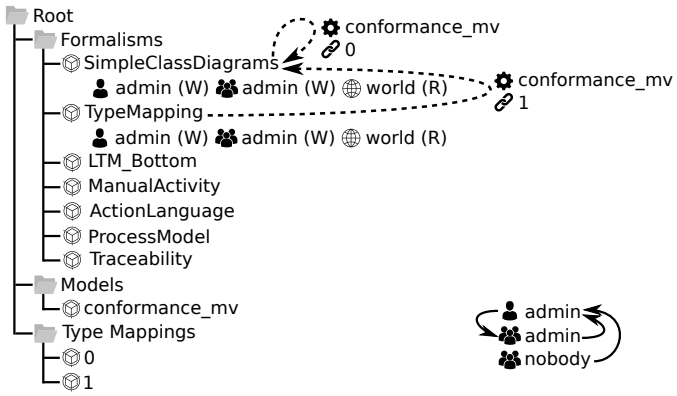


Figure 5.7: Initial bootstrap FTG including metadata and access control (excerpt). Instance-of links are parameterized with a conformance semantics (e.g., *conformance_mv*) and type mapping model (e.g., *0*).

megamodel can also be used to store meta-data, which is stored in additional attributes. We have provided support in this megamodel to store information about users, groups, model hierarchy, and traceability links.

5.5.2 Model

Figure 5.7 presents an excerpt of the initial FTG that is loaded when the Modelverse is initialized, and thus without any non-critical model. Only an excerpt is shown for readability, as even at initialization a lot of meta-information is stored. Full information is shown only for the SimpleClassDiagrams formalism, to give an indication of which information is stored. This model is hierarchically contained in the folder "Formalisms", which resides in the root of the hierarchy (folder "/"). It has the following access permissions: the owner (user "admin") has read/write permissions, users that are member of the owning group (group "admin") have read/write permissions, and all other users (world) have read-only permissions. The model conforms to itself (the self loop) and for this relation is parameterized with "conformance_mv" conformance semantics (Section 5.6), and the type mapping is stored as a separate model (stored as model "0" in the "Type Mappings" folder). Another example is the "Type Mapping" formalism, which conforms to "SimpleClassDiagrams". Note that a single model might conform to multiple metamodels, or even to the same metamodel with a different mapping or semantics, which is why conformance associations can be parameterized. The same information is stored for each model in the Modelverse.

Besides the models and their hierarchy, the FTG also stores user information: all user data (e.g., name and password) is stored, as well as the groups they belong to and which they own. Each model has an owning user and group, which was mentioned with the access permissions.

This model is updated as the Modelverse is executing, thereby adding or removing models, conformance relations, and users. All changes can be performed through the use of the usual model management operations, as the FTG is itself a model. This also means that it is possible at runtime to introduce new (types of) relations.

5.5.3 Evaluation

We now briefly evaluate the model of the Formalism Transformation Graph with respect to our original motivation. No detailed evaluation is given for this topic, as this does not present a significant contribution.

By storing all meta-information explicitly, in the form of an extended FTG, we provide simple access to all information of these models. As the FTG is a model itself, it can be queried and manipulated as any other model. Besides, no data structure needs to be defined to store the models and their meta-information: everything is explicitly modelled and available to users. This was seen in the core library, where all operations that add or remove models are merely instantiating or removing elements from the FTG. As soon as the FTG is updated, this change is visible to all users simultaneously.

It is also possible for models to have multiple conformance links, which are annotated with additional information pertaining to the conformance relation. Again, by updating the associations in this model, the actual relations between the models change. By storing for each element in the FTG a timestamp when it was last modified, it would be possible to cache results: if the conformance relation was created after the latest modification of the model, we can guarantee that the model still conforms without doing the conformance check again.

The same model is used to manage all users and groups, and again all such operations become merely changes on the FTG model. Due to the high amount of information stored in this model, and its sensitivity (e.g., user password hashes), it is best set to be inaccessible for all users except for the admin users.

5.5.4 Link to Requirements

Explicitly modelling the FTG, used for model management and meta-data, has an influence on **Requirement 1 (Language Engineering)**, **Requirement 2 (Activities)**, **Requirement 3 (Process Modelling)**, **Requirement 4 (Multi-user)**, and **Requirement 8 (Access Control)**.

Requirement 1 (Language Engineering) is influenced due to the megamodelling involved. Various languages can have relations between one another, such as traceability when the current language is actually the merger of multiple other languages. Like everything else, these inter-model links are modelled explicitly in the FTG as well, making it possible to operate with them. In the future, fragment-based composition of languages, partially enabled by the FTG, might have further effects on language engineering.

Requirement 2 (Activities) is influenced because of the explicit activity signature stored in the FTG. For each activity, the set of input and output models is specified in this megamodel of all models. This makes it possible to search applicable activities for a specific model, or vice versa search for all possible models that can be passed to an activity.

Requirement 3 (Process Modelling) is influenced because the FTG part of the FTG+PM can be automatically constructed based on the process model, filtering models from the single big FTG.

Requirement 4 (Multi-user) is influenced, as the FTG can be used for multi-user collaborations as well. For example, it can be used to store version control information (e.g., model

history, revision number, author, log information) and locking information (e.g., has a user locked a model, does the user have an exclusive lock). While these are not implemented, the use of an FTG provides many future opportunities in this direction.

Requirement 8 (Access Control) is influenced, as access control is implemented through the FTG. Indeed, for each model some metadata is stored in the megamodel, which can later on be used for access control. The same megamodel is also used for navigation through a tree structure, thereby also allowing access control on the directories grouping models together.

Summary

The FTG references the various formalisms, models, and activities stored in the Modelverse. By explicitly modelling the FTG as a megamodel, it becomes possible to reuse existing model management operations on the core data structure of the Modelverse. Additionally, it can be used to store meta-information of models, such as access control information, traceability links, and model versioning, and of users, such as passwords and group membership.

5.6 Conformance Algorithm

The conformance algorithm defines what it means for a model to conform to a metamodel. To do this, it has to provide the semantics associated with the meta-meta-model, such as the *Class Diagrams* formalism. For example, what does the inheritance link mean, is multiple inheritance supported, which attributes are supported on classes, what does the multiplicity value do, and so on. The importance of this algorithm can thus hardly be overestimated, as it forms the foundations of any (meta-)modelling tool [32, 41, 89, 279]. In this section, we focus on the linguistic conformance relation, being the primary dimension of interest to modellers and language engineers.

5.6.1 Motivation

For the linguistic conformance relation, many diverging implementations are given in the literature, where none agree on what it means for a model to conform to a metamodel. For example, AToMPM [273] and MetaDepth [79] both support multiple inheritance, but they have different precedence when attributes are in conflict. Even similar tools, by the same authors, are often incompatible: AToMPM [273] and AToM^3 [86]; WebGME [191] and GME [175]; DPF [173] and WebDPF [228]. Even different versions of the same tool can be incompatible, either intentionally [89], or accidentally due to (minor) implementation changes or bugs. This makes meaningful model exchange between these two tools (or even two versions of the same tool) difficult. This will be our primary motivation for explicitly modelling the conformance relation. Most of the time, there is even no clear definition of conformance for any of these tools: details are implemented in the source code, making it impossible to understand without reverse engineering or trial and error.

This raises the question as to what "instantiation" and "conformance" mean in most tools: each tool implements its own algorithms, based on what they believe that it means [29, 84, 99, 131, 266, 267]. The reason for this is simple: even the UML [5] doesn't state what it

means for a model to be instantiated or conform to another model [14, 17, 30, 31]. Some tools merely document their algorithm as "following long established semantics" [30, 323], without being more explicit about it. Many variants exist [170, 230], and the meaning of conformance can even vary between different layers in the meta-hierarchy [31]. It should thus not come as a surprise that the exact meaning will still take some time to settle [41], while each tool implements its own variation [88].

The problem gets even more obvious when multi-level modelling is added to the mix. Being a relatively new field that is still expanding, many new concepts are being introduced, such as potency. Interpretations of potency vary widely [239]: there is normal potency [169], leap potency [81], and potency for associations [25].

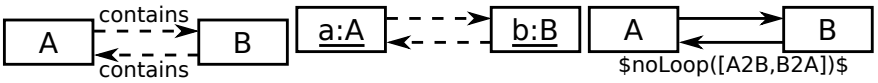
Another example is the reliance on hardcoded associations, which are not checked by the conformance algorithm, but by the implementation. For example, many tools hardcode inheritance links [175, 191, 235, 273], conformance relations [67, 90, 99, 100], aggregation and composition links [15, 19, 295], containment links [177], multiplicities [40, 177, 279], or even version numbers [142]. By not checking these values explicitly, but relying on the underlying implementation, tool semantics becomes unclear.

Similarly, most tools hardcode their meta-circular level [17, 237, 243, 273]. This makes it impossible to use another Model at the Meta-Circular Level (MMCL) than the one(s) for which the type/instance relation was defined.

Due to these intentional or accidental differences, inconsistencies between tools arise: all models, including languages, become grafted on the tool's implementation. With the growing importance of tool interoperability, this turns into a problem: while models can be exchanged, inherent tool semantics cannot. This problem is particularly relevant for model repositories, as there is no universally agreed upon MMCL, nor an agreed upon set of physical links [36, 98, 191], nor an agreed upon conformance algorithm. Without this, model repositories are forced to make decisions on which models they accept and process. An extreme case is ReMoDD [113], where models are merely treated as files, without attaching any semantics, thereby quickly reducing its use to that of a file server. Another case is MDEForge [36], where models are first-class citizens and are fully supported, for example through model transformations [99]. Distil [190] is another approach, which provides a DSL and rich tool support for both the storage and service definition part of the service, including code generation and deployment.

We distinguish two types of variations between tools: syntactical and semantical. Syntactical variations result in non-exchangable models, and semantical variations cause unexpected behaviour with successfully exchanged models. A simple example is multiple inheritance. Syntactically, some tools do not support this, making them unable to receive models from tools which do support multiple inheritance. Semantically, some tools handle the resolution order of multiple inheritance differently, thereby altering the set of allowed instances.

We tackle this problem by explicitly modelling the language syntax and semantics (i.e., conformance), usually built into the tool. Apart from serving as documentation, new syntax and semantics can then be loaded on-demand without altering the tool. As such, a single tool is able to store and operate *semantically meaningful* models of different tools, given that explicit models are present. Models are then not grafted on tools, but on other models, which can just as well be exchanged.



(a) AToMPM meta-model. (b) Invalid instance. (c) Metadepth meta-model.

Figure 5.8: First example language: use of containment links.

To concretize the problem, we focus on two rather similar tools: AToMPM [273] and Metadepth [79]. For both tools, we describe some (hardcoded) differences to illustrate the problem. We present minimal example languages and models for both syntactical and semantical differences.

Syntactical Variations

First are syntactical variations, caused by a different abstract syntax of the meta-language. Such changes can automatically be detected, as a model would rely on unknown constructs. We show two examples: one with a feature of AToMPM that Metadepth does not support, and one that is the other way around.

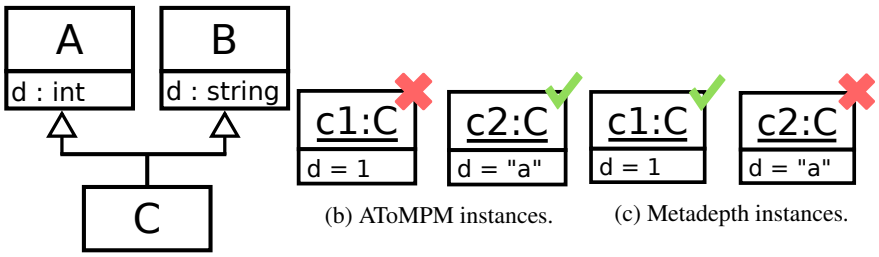
The first example language, in Figure 5.8a, uses a specific kind of association: the containment relation, as supported by AToMPM. It resembles an ordinary association, but indicates that the source element is a container for the target element. Its primary use is for visual representation, though it is also used as implicit constraint: containment cycles are not allowed. Instances of class *A* can contain instances of the class *B*, and the other way around. Figure 5.8b shows an example instance of this language, where *a* contains *b*, but also the other way around. Conceptually, this does not make any sense. With a containment relation, this is automatically flagged as an error and the model does not conform. In Metadepth, which does not support a containment relation, this same language cannot be loaded; the association type "containment" is unknown. As such, the model cannot be exchanged either, as it depends on the language. It is possible to mimic the containment relation in Metadepth by defining the containment relation as a normal association, which has an additional constraint that does not allow loops. A semantically equivalent meta-model is shown in Figure 5.8c.

The second example language, in Figure 5.9a, uses multiplicities on a class, as supported by Metadepth. The lower and upper cardinality is defined as an integer attribute on the class. Its primary use is to restrict the number of instances of this specific type: the number of instances must be within this range. The class *A* requires that there are exactly two instances of *A* in every model conforming to it. Figure 5.9b shows an example instance, where only one instance of *A* is present. With the class multiplicities, this is automatically flagged as an error and the model does not conform. In AToMPM, which does not support class multiplicities, this same language cannot be loaded: the attribute "multiplicity" is unknown. It is possible to mimic multiplicities in AToMPM by defining a global constraint, which checks the number of instances of *A*. A semantically equivalent meta-model is shown in Figure 5.9c.

In both example languages, the tools are equivalent in their expressiveness (i.e., they can be used to express the same language), but the language must be represented differently. As such, languages, and therefore models, cannot be easily exchanged without a conversion at the abstract syntax level.

(a) Metadepth meta-model. (b) Invalid instance. (c) AToMPM meta-model.

Figure 5.9: Second example language: use of node multiplicities.



(a) Metamodel. (b) AToMPM instances. (c) Metadepth instances.

Figure 5.10: Third example language: use of multiple inheritance.

Semantical Variations

Second are semantical variations, caused by a difference in the implementation of the conformance check, which provides the semantics for the abstract syntax of the meta-language. Clearly, just calling an association "containment" or an attribute "multiplicity" does not automatically give it the correct semantics: it needs to be defined somewhere. Semantic differences are indetectable when models are exchanged, as they structurally conform.

Note that we consider the semantics of the meta-modelling language (i.e., what does a given meta-model mean), and not the semantics of the modelling language (i.e., what does a given model mean). The former is mostly hardcoded in the tool, whereas the latter is domain-specific and implemented using, for example, model transformations.

The third and final example language, in Figure 5.10a, uses multiple inheritance, as supported by both AToMPM and Metadepth. An example of such a language is shown in Figure 5.10a, where the class C inherits from both A and B. Both A and B define the same attribute but with different types. It is unclear which of the two is selected for C, which inherits from both. The semantics attached to multiple inheritance, responsible for the choice, is hardcoded in both tools and left undocumented. Only experimentation is therefore possible to figure out what it means, resulting in Figure 5.10b for AToMPM and Figure 5.10c for Metadepth. As the set of conforming instances differs, for the same language, both tools attach different semantics to the language. AToMPM seems to resolve the earliest created inheritance link, whereas Metadepth seems to lexicographically sort the class names and picks the first match.

The semantics of the inheritance relation is one of the well acknowledged semantical variations in tools. While many tool developers claim that they "just use inheritance", there are actually a lot of variations to how it has to be interpretted [81, 90]. For example such as in SELF [283], similar to Java [96], using prototypical inheritance as in GME [175] and

WebGME [191], or customized [82]. Support for multiple inheritance, and specifically the resolution order, is frequently left undocumented, though it should be made explicit [202]. Many tool even define new inheritance links between elements, for which there is absolutely no pre-existing notion of what it means for them to inherit from each other. Examples can be found in fragments [15], concrete syntax [22], packages [14], mixin layers [80], and even in statecharts [134].

Making explicit the notion of what all of this means was already done previously [19, 279], but only formally: the actual implementation does not yet give a hint to the user as to what it means. Users wanting to understand, will need to either read the (very formal) documentation, or look at the implementation of the tool itself. As those users that will be using the tool are likely to be domain experts, and not formal computer scientists or programmers, this will not aid these users in understanding what they are doing. The exception to this is Maude [67], where the conformance relation is explicitly modelled within Maude itself [235]. Maude, however, does not tackle the problems identified further in this paper.

While the example difference here is likely intentional, many other differences exist that are likely accidental (e.g., bugs or ommissions). For example, AToMPM does not check the type of attributes, and Metadepth cannot connect edges when inheritance is involved, nor can it have attributes with specific keyworded names (e.g., “id”). Notwithstanding the source of the difference, the semantic differences make model exchange meaningless. And when the tool semantics is altered (e.g., an intentional change, a bugfix, or a newly introduced bug), it is possible that a previously conforming model suddenly becomes invalid, or that a previously invalid model suddenly becomes valid.

These variations give rise to the “meta-muddle” [109], which prevents users from understanding the finer details of what they are actually doing. Actually, having all of these different semantics around isn’t that bad, as it is only natural in a relatively new research domain. Similar fragmentation existed in the beginning of Object Orientation, with many variations being developped in parallel [331]. However, the notion of what a model type is, isn’t widely agreed upon [235], and in the current state of affairs, users will only find out what it means through trial and error.

Problem Scope

The previously introduced problems are often irrelevant for users of a single tool. As such, we do not propose to alter existing (meta-)modelling tools, though it might be useful to have sufficient documentation. The true problem lies elsewhere, with tools that explicitly have to communicate with many different tools: model repositories, such as the Modelverse.

The primary purpose of model repositories is model storage and exchange. It makes sense that they want to maximize the set of supported tools, thereby maximizing the available models. Model repositories, however, have to understand the models they are managing, as otherwise they would be reduced to a mere file server. When working with models from different tools, and possibly exchanging them between these tools, it becomes important to take these syntactical and semantical variations into account.

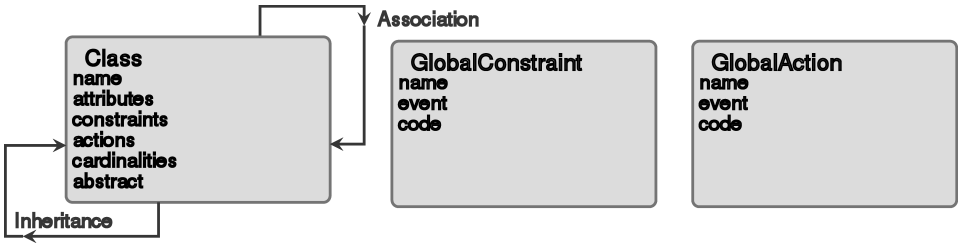


Figure 5.11: AToMPM's M_3^{AS}, taken directly from AToMPM itself.

5.6.2 Model

Varying abstract syntax and semantics at the meta-language level were identified as the root of the problem. Current tools acknowledge that an explicit meta-model is required to create a flexible modelling tool, in which the language can be altered. They do not, however, take this one level up the meta-modelling hierarchy: the language used to create new languages, the meta-language, is hardcoded in the tool. Being hardcoded, the problem becomes even worse: they are not flexible either, as the hardcoded aspects cannot be altered in any way. For this reason, we propose to explicitly model both the meta-language's abstract syntax and semantics, and make this fully flexible at runtime.

Some aspects of current tools are already modelled explicitly. We distinguish three layers, as commonly agreed upon: *M1* (model level), *M2* (language level, or meta-model), and *M3* (meta-language level, or meta-meta-model). For Metadepth, both *M1* (M_1^{AS}) and *M2* (M_2^{AS}, M_2^{SEM}) are explicitly modelled. In AToMPM, the syntax of *M3* (M_3^{AS}) is additionally explicitly modelled. Both tools hardcode the semantics of *M3* (M_3^{SEM}). It is this aspect of the tool that we will also model explicitly in our approach.

As both M_3^{AS} and M_3^{SEM} are explicitly modelled in our approach, it becomes possible to (1) alter them at runtime (e.g., optimizations, refactorings); (2) create and use new ones at runtime (e.g., support a new tool, bugfixes); and (3) have multiple of them simultaneously (e.g., models from different tools loaded in a single tool).

Meta-meta-model (M_3^{AS})

The meta-meta-model M_3^{AS} defines the concepts that can be used when defining a new language, or meta-model. It has two primary purposes. First, it can serve as documentation for language engineers: what is the name of attributes, what constraints can be added, whether multiple inheritance is supported, and so on. Second, it is required for several operations that need an explicit meta-model. For example RAMification [171], used to create a new language to express model transformation rules by Relaxing, Augmenting, and Modifying the existing language. Differences in M_3^{AS} lead to syntactical differences between tools, which can be automatically detected by comparing two M_3^{AS} models.

The M_3^{AS} of AToMPM is shown in Figure 5.11, modelled explicitly using Entity-Relation Diagrams. It shows the various attributes that can be set on a class, such as "attributes" to define new attributes, and "name". Perhaps surprisingly, attributes have no dedicated entity, in contrast to other approaches, such as EMF.

For the Modelverse, the used M_3^{AS} is shown in Figure 5.12.

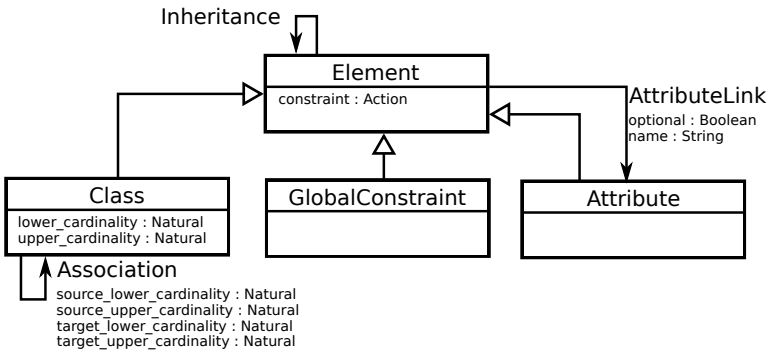


Figure 5.12: Modelverse's M_3^{AS}.

Meta-Language Semantics (M_3^{SEM})

The meta-language semantics M_3^{SEM} defines the semantics of concepts defined in M_3^{AS}. It takes a model and its meta-model as input, and determines whether the model conforms to the meta-model. Its primary purpose is in determining whether a model is valid with respect to a given language specification (i.e., check conformance). Differences in M_3^{SEM} lead to semantical differences between the tools, which are difficult to detect automatically. Indeed, we would need to verify if two models for M_3^{SEM} behave exactly the same in every possible context.

A snippet of the conformance algorithm of Metadepth is shown, pertaining to the multiplicity checks of classes in Algorithm 3. This pseudo-code can be modelled in an explicitly modelled action language.

ALGORITHM 3: M_3^{SEM} snippet for the cardinality check.

```
for all class ∈ allInstances(M3, Class) do
    assert allInstances(M2, class) ≥ class.lower_cardinality
    assert allInstances(M2, class) ≤ class.upper_cardinality
end for
```

For the Modelverse, the used M_3^{SEM} is shown in Algorithm 4. There are checks for the following characteristics: 1) whether the element is typed at all; 2) whether the elements type is part of the desired metamodel; 3) if the element is an edge, we check that both source and target conform to the source and target of the type; 4) for each outgoing edge in the type, we check whether the number of instances for the element is correct with respect to the cardinalities in the type; 5) the same happens for all incoming edges; 6) we evaluate the local constraint function for this element (if applicable); and 7) evaluate the global constraint functions on the complete model. As soon as one of the tests fails, the model is deemed non-conforming, and this error message is returned.

5.6.3 Evaluation

As a proof of concept, we implemented this approach in the Modelverse, which can now be extended with new conformance algorithms (M_3^{SEM}) and meta-circular levels (M_3^{AS}).

ALGORITHM 4: M_3^{SEM} pseudo-code for the Modelverse.

```
for all e ∈ M do
    if e ∉ TM then
        return "Element has no type specified"
    end if
    if TM(e) ∉ MM then
        return "Type of element not in specified metamodel"
    end if
    if is_edge(e) then
        if not is_nominal_instance(edge_src(e), edge_src(TM[e])) then
            return "Source of model edge not typed by source of type"
        end if
        if not is_nominal_instance(edge_dst(e), edge_dst(TM[e])) then
            return "Source of model edge not typed by destination of type"
        end if
    end if
    for all te ∈ outgoing(tm[e]) do
        if count(outgoing(e, te)) > read_attr(te, "target_upper_cardinality") then
            return "Target upper cardinality violated"
        end if
        if count(outgoing(e, te)) < read_attr(te, "target_lower_cardinality") then
            return "Target lower cardinality violated"
        end if
    end for
    for all te ∈ incoming(tm[e]) do
        if count(incoming(e, te)) > read_attr(te, "source_upper_cardinality") then
            return "Source upper cardinality violated"
        end if
        if count(incoming(e, te)) < read_attr(te, "source_lower_cardinality") then
            return "Source lower cardinality violated"
        end if
    end for
    constraint_result ← read_attr(e, "constraint")(m, e)
    if constraint_result ≠ "OK" then
        return constraint_result
    end if
end for
for all cs ∈ all_instances(MM, "GlobalConstraint") do
    constraint_result ← read_attr(cs, "constraint")(m)
    if constraint_result ≠ "OK" then
        return constraint_result
    end if
end for
```

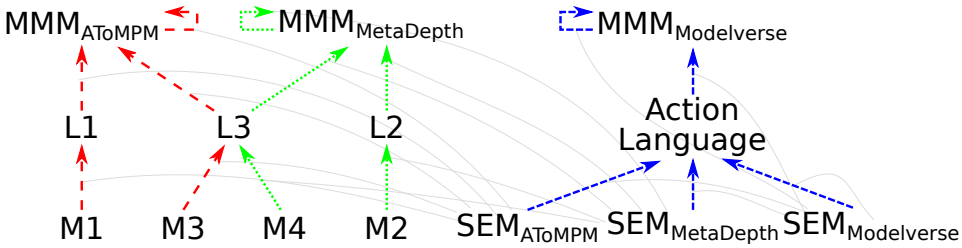


Figure 5.13: Overview of languages, models, and relations in the Modelverse.

When defining a new conformance relation, this is a model in the Modelverse and an extension to the Modelverse at the same time. The interrelations between all models, such as conformance relations, are also modelled explicitly. For each conformance relation, the link is specified by the source model (instance), target model (meta-model), and conformance semantics. As such, a model can conform to the same meta-model through multiple conformance semantics, or to different meta-models using the same conformance semantics, or to different meta-models using different conformance semantics.

Using this explicitly modelled approach, the Modelverse is able to offer the same results for the conformance checks as either AToMPM or Metadepth, or any other tool, for that matter. An overview of how these explicitly modelled interrelations are stored, is shown in Figure 5.13, which itself represents a model in the Modelverse. While *L1* and *L2* can only be created in AToMPM and Metadepth, respectively, *L3* is a valid language in both. Nonetheless, both tools attach a different semantics to *L3*, as seen in the two different instances: *M3* and *M4*. Both conformance semantics are also explicitly modelled, as shown at the right. These are again typed by something, in this case using a conformance relation defined specifically for the Modelverse, though also explicitly modelled. This could just as well have been any other previously defined semantics.

The Modelverse can therefore be used to host a variety of models and languages from different tools, potentially having different syntax and semantics. Although it is of course required to model M_3^{SEM} and M_3^{AS} of the tool the model originates from.

5.6.4 Related Work

The conformance relation plays a crucial role in Model Driven Development (MDE) [32, 279]. But while it has often been studied, it has remained hardcoded in various tools. Research up to now has mostly focussed on defining different levels of conformance [156] and new types of conformance, such as relaxed [245] or partial [264]. Nonetheless, they are hardcoded and partially inflexible as well: tools cannot be extended with additional conformance relations, nor can existing conformance relations be inspected or manipulated.

Updates to the hardcoded syntax and semantics, such as UML, have resulted in (often unnecessary) breakage of conformance [89]. As such, old models cannot be loaded in newer versions of the same tool, if tools update their implementation of the UML. But whereas a model might still load in a newer version of the UML, nothing guarantees that the same semantics are used in its evaluation.

In the multi-level modelling community, many aspects of conformance are still being investigated, specifically for newly introduced physical attributes. Since it is still very much a research domain, new additions to the paradigm are introduced at a steady rate. As the definition of potency was vague, and even in conflict with each other in [27, 81, 82], many variations of multi-level modelling currently exist [23, 90]. And as we don't claim that any is better than the other, all of them has its own reasons to exist.

Problems begin, however, when seperate paradigms start to introduce additional of these structural constraints. For example leap potency [81], deep multiplicities [25], deep constraint languages [24], mutability [22], durability [22], and dual fields [29]. Each of them has a non-precise semantics, which is mostly hardcoded inside of the tool implementation. These concepts are thus not easily portable between different paradigms, or even between different tools. Even worse, as these constraints are defined independently, is the potential interplay when they are combined [79]. Some of these constraints, like potency, have already been found out to have a significant impact on all other aspects of (meta-)modelling, such as model transformations [21, 84, 323]. And since many of these constraints are only supported by one or so tool, model interoperability is lost completely, requiring models to be written out in a different form that encodes the same information, but without these special constructs [130].

When models are exchanged, the first thoughts are of the technological problems: how to transfer the data from one tool to another. Various serialization formats were conceived for this problem, such as XMI and JSON. Nonetheless, they limit themselves to transfering data only, not the actual model. Essentially, M_2^{AS} is exchanged, but M_3^{AS} and M_3^{SEM} are not. As such, models can be exchanged, assuming that both tools implement the same M_3^{AS} and M_3^{SEM}. Should they differ, the exchanged data becomes semantically meaningless.

This brings us to model repositories, which, using this approach, often resort to semantically meaningless model exchange. For example, ReMoDD [112], being as general as possible, sacrifices model semantics: uploaded models have only marginally more semantics than arbitrary files. Advanced operations, which rely on the semantics of these models, are not supported. Another solution, as taken by MDEForge [36], is to restrict exchanged models to its own M_3^{AS} and M_3^{SEM}. This allows them to actually use the models, for example for model transformations [99], though they do so by limiting the set of supported tools.

A posteriori typing [83] has been proposed as a way to have multiple types for a single model. But whereas our approach does not consider any conformance relation as special, a posteriori typing starts from a special relation: the constructive type. The constructive type is the type used to instantiate the model, and cannot be changed. All other types, discovered on-the-fly, are completely flexible. Even though multiple meta-models can be found for a single model, all conformance relations must use the same M_3^{AS} and M_3^{SEM}. As such, only M_2^{AS} can change, and not M_3^{AS} nor M_3^{SEM}, though this provides sufficient knowledge to reuse operations between different meta-models. Concepts [80] serve a similar purpose, but also with similar restrictions: there is only freedom at M_2^{AS}.

Another problem, which we did not tackle here, is which constraint language is used. While there are some standards defined, such as OCL [3] or EOL [164], many tools deviate from these to allow for their own special constructs. Examples are MetaDepth's extended EOL [81] and ETL [84] and merging with Java [79], Melanie's deep constraint language [24], Nivel's Weighted Constraint Rule Language [19], and GeMoC's Event

Constraint Language [174]. So not only the instantiation and conformance semantics are incompatible, but even the used constraint languages don't match with each other.

5.6.5 Link to Requirements

Explicitly modelling conformance, and allowing for multiple conformance relations simultaneously, has an influence on **Requirement 1 (Language Engineering)**, **Requirement 2 (Activities)**, **Requirement 7 (Model Sharing)**, and **Requirement 10 (Portability)**.

Requirement 1 (Language Engineering) is influenced due to the reliance on meta-model constructs. A language engineer often makes use of specific constructs that are known by the conformance check, such as inheritance, multiplicities, potencies, and so on. The semantics of these concepts, however, is often only defined in the tool implementation

Requirement 2 (Activities) is influenced, as the multiple conformance relations can have an influence on which "view" on the model is taken during the activity. For example, when a model conforms to multiple metamodels, the correct meta-hierarchy of this model is automatically selected based on the signature of the activity.

Requirement 7 (Model Sharing) is influenced, as sharing models between tools requires the presence of an explicitly modelled meta-circular level and conformance relation, to be able to "mimic" the other tool.

Requirement 10 (Portability) is influenced as most of this semantics is normally hard-coded in the tool, making porting difficult, as all the semantics must be identically copied and reimplemented. By modelling this explicitly in a neutral language, such as the Action Language, these operations become portable across different platforms.

Summary

The conformance algorithm is of utmost importance in any (meta-)modelling tool, as it is used to manage the meta-hierarcy. It is, however, hardcoded in most tools, resulting in different implementations, and therefore incompatibilities between tools. These incompatibilities, both syntactical and semantical, result in difficulties with meaningful model exchange: models are grafted on the tool's implementation. By explicitly modelling all aspects of the conformance relation, both abstract syntax and semantics, models become grafted on another model, which can also be exchanged. Furthermore, a single model can conform using multiple such semantics, potentially to different abstract syntax definitions as well.

5.7 Physical Type Model

Another dimension of the conformance relation is the physical dimension. The physical dimension defines how models are physically represented on the implementation platform, such as Python or Java. We call the metamodel in this dimension the Physical Type Model (PTM). Possibly, models can be stored in a relational database, in which case their physical type model resembles, for example, SQL.

Ideally, the modeller is unaware of the physical dimension. The language engineer and Modelverse developer, however, often require access to the physical level. For the language engineer, some operations might be generic and applicable to multiple languages simultaneously, in which case a more generic means of access is required. For example, querying all edges does not depend on the language that is being used, and should therefore be done at the physical level. For the Modelverse developer, most of the core library functions are implemented in a language-independent way, often crossing multiple levels in the meta-hierarchy. For example, the conformance algorithm naturally spans two languages, the instance and type model, and is therefore implemented at the physical level. By having these users rely on the physical type model, changes to the physical type model are not allowed, as they would break many operations.

5.7.1 Motivation

One of the shortcomings of current (meta-)modelling tools is their strong reliance on the implementation level. This reliance ranges from exposing the general purpose implementation language used (*e.g.*, Java), to requiring some operations to operate directly on the internal data representation of the models (*e.g.*, XMI). Tools voluntarily chose for this strong reliance for several pragmatic reasons: the implementation language and internal data representation already exist and are sufficiently mature. Additionally, the used algorithms (e.g., for model management) are complex and often operate at a lower level than other operations: they need access to the internal data representation [15, 273]. We consider some of these problems in more detail.

First, tool developers are familiar with some mature programming language, which has extensive tool support, such as efficient compilers, debuggers, and code analyzers. Additionally, many libraries are available for use, such as graphical libraries, parsers, and data structures. It is therefore logical that they wish to reuse as much of this as possible, thereby becoming more and more dependent on this language and its features.

Second, explicitly modelling generic algorithms is non-trivial, as they often span multiple levels in the meta-hierarchy. An example is shown in Figure 5.14, where a petri net model linguistically conforms to a simple petri net metamodel. A conformance algorithm needs access to the metamodel (*e.g.*, *Place*) and the model (*e.g.*, *a place*), to check whether they conform. The conformance algorithm itself, however, needs access to both layers, as shown in Figure 5.14, which is impossible in a strict metamodelling hierarchy. As noticed in the Figure, this strictness violation is not present when going to the physical dimension (i.e., to the implementation).

Third, the conformance algorithm is generic and applicable to more domains than only Petri Nets. As such, implementing it for Petri Nets concepts only is wasteful. When implemented in the physical domain, generic concepts can be used (such as node and edge), thereby making it applicable to all domains.

Manipulations at this lowest level are not easy to model, as the physical level is often obfuscated itself. For example, megamodelling [43, 44] often requires additional links between models (e.g., traceability, inheritance, materialization), which the UML leaves implicit [247]. The semantics of these links is thus also ported to the physical level, making it difficult to reuse existing database systems, since the database has to be augmented with

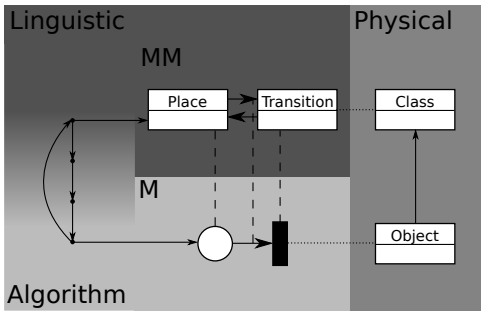


Figure 5.14: An algorithm spanning multiple layers in the linguistic dimension, violating strict metamodelling.

meta-modelling concepts [274]. By not reusing existing database systems, it becomes impossible to reuse advances in this research domain.

The disadvantages are significant, as algorithms become grafted on the tool, just like conformance was grafted on the tool's implementation. We consider two disadvantages: (1) the tool implementation must remain fixed and (2) it is difficult to reuse algorithms from other tools.

The tool implementation must remain fixed, as algorithms intimately depend on the internal API and the details of the data structure. Should, for example, the internal data representation be changed ever so slightly (e.g., add a new physical attribute), all generic algorithms become incompatible. Similarly, if the tool is ported from one implementation language to the other (*e.g.*, from Python to C, for efficiency), all Python code has to be ported to C as well.

Reusing existing algorithms also becomes difficult, thus hindering portability and reuse. This is especially important in the scientific community, where new or updated algorithms are continuously introduced to further the state of the art. These algorithms then strongly rely on the internal representation of models, such as XMI or graphs. Implementing these algorithms in a tool with a different internal representation proves challenging due to the different assumptions that were be made and the different API of this data structure. For example, an algorithm that assumes it is working on a graph-like model is non-trivial to implement on a model that is stored in a relational database.

To counter these disadvantages, we combine the best of both worlds: we explicitly model the PTM in the linguistic domain, thereby shifting all physical operations to linguistic activities. This makes such activities portable between different tools, and independent of the actual PTM that is implemented. In essence, this is similar to explicitly modelling the conformance relation: the physical representation of models becomes grafted on another model, instead of on the tool's implementation.

5.7.2 Model

Recall that relying on the physical type model was the main cause of the problems we have previously observed. To counter these problems, we undertake several steps: we copy the

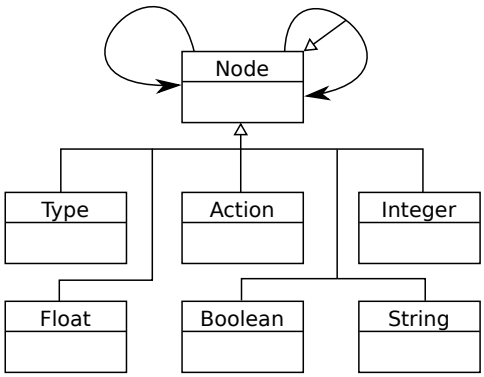


Figure 5.15: $LTM_{\perp}$, allowing for any element to connect to any other element.

PTM to the linguistic domain, use it in a multi-conformance setting, and then shift existing physical operations to this dimension.

Moving Away from the Physical Type Model

There is a natural relation between both physical and linguistic metamodels, as both are related to the structure of the model. To do this, we define a new metamodel, which is identical to the (implicit) metamodel of the implementation layer (the PTM). This metamodel, however, is defined in the linguistic dimension, thus making it explicit. For clarity in our discussion, we call this metamodel $LTM_{\perp}$, shown in Figure 5.15. It can be seen that it is a metamodel for basic graphs, where nodes might have values. These possible values are *Type* (the type of any value type, including itself), *Action* (the type for all action language constructs, such as *While*, *If*, and *FunctionCall*.), *Integer*, *Float*, *Boolean*, and *String*. Additionaly, edges are a subclass of nodes, meaning that they can have incoming and outgoing edges themself. Since every element is a subclass of *Node*, an edge can start and end at any element, including itself. As this is only at the conceptual level, it was done to make reasoning about edges from edges conceptually clearer. The leftmost association from *Node* to itself represents the type of inheritance relations: since inheritance relations are also explicitly modelled [314], they require their own metamodel. And since the $LTM_{\perp}$ should be self-describing, it contains this type too.

Since any model conforms to the (often implicit) physical metamodel in the physical dimension, they should also, by definition, conform linguistically to $LTM_{\perp}$. We call this new linguistic conformance to $LTM_{\perp}$ $conformance_{\perp}$. While it is actually the same as conformance in the physical dimension, we shift this to the linguistic dimension to offer it to the users. Thanks to the possibility for multiple metamodels for a single metamodel [314], it is possible for the model to be typed by multiple linguistic metamodels: $LTM_{\perp}$, and the original linguistic metamodel(s). Figure 5.16 shows the 1-to-1 mapping of the PTM to the linguistic dimension. As each element necessarily conforms to the PTM, it will also, by definition, conform to the new $LTM_{\perp}$.

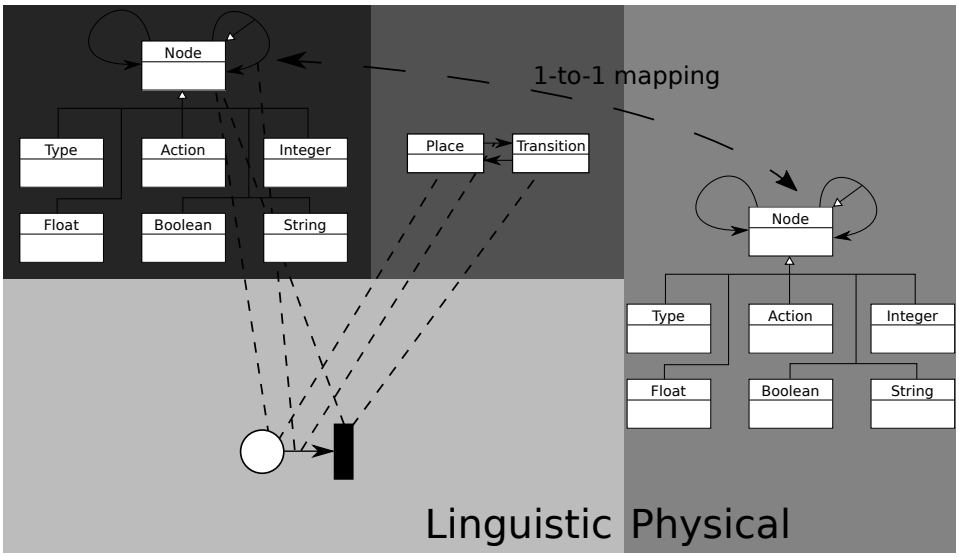


Figure 5.16: $LTM_{\perp}$ added in the linguistic dimension, which is identical to the one in the physical dimension.

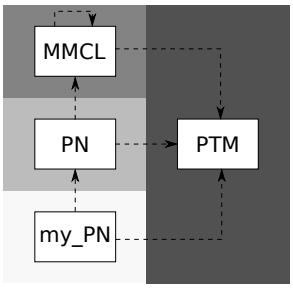
Coping with Strict Metamodelling

By lifting the physical conformance relation up to the linguistic conformance dimension, we achieve a way of explicitly modelling, albeit indirectly, in the physical dimension. Users are therefore able to, using their normal linguistic modelling tools, alter the physical dimension. The physical representation of the model is thus seen as an instance of a linguistic metamodel.

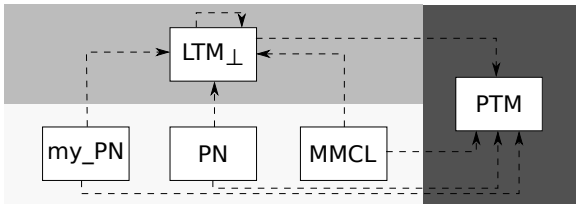
While the tool still complies to strict metamodelling in the linguistic dimension, $LTM_{\perp}$ is taken so general, that the complete metamodelling hierarchy can be expressed as a direct instance of it. This effectively flattens the original metamodelling hierarchy into a single level: $LTM_{\perp}$ at the metamodelling level, and everything else at the modelling level. In this single model level, which is only a different view on the same model, strict metamodelling does not restrict anything, even links between different levels (of the original hierarchy). Figure 5.17 represents the two possible views on the modelling hierarchy: either through the usual conformance relation (Figure 5.17a), or the new $conformance_{\perp}$ relation (Figure 5.17b).

Depending on the used metamodel and conformance relation, strict metamodelling can thus be interpreted differently. Note that this is still distinct from dropping strict metamodelling completely: strict metamodelling is still used throughout the complete environment, and still imposed on instances, even with the $conformance_{\perp}$ relation. But the implications of strict metamodelling depend entirely on the metamodel: for normal linguistic metamodels, strict metamodelling is as it was originally designed, but for the special metamodel $LTM_{\perp}$, strict metamodelling does not constrain anything because every element is at the same level.

Coping with strict metamodelling alone does not solve all problems. While the limitation



(a) Petri Nets metamodel



(b) $LTM_{\perp}$

Figure 5.17: Different modelling hierarchies for the model *my_PN*, as seen through two different linguistic views.

of not being able to model executable models across levels was removed, these executable models still directly interact with the underlying data structure. This is still a lingering aspect of the physical dimension, which we tackle next.

Abstracting Implementation Details

The 1-to-1 mapping between the physical metamodel and $LTM_{\perp}$ made it possible to linguistically access the physical dimension. But the physical dimension is still part of the implementation, and could therefore change in subsequent versions. This would bring us to language evolution, as $LTM_{\perp}$, and possibly $conformance_{\perp}$, would also have to be updated, together with all saved models. While some advances are made to language evolution in order to do these changes automatically, we don't want to expose users to these problems.

Users should therefore not be bothered with the internals of the tool, not even the physical data representation. And while users do need access to a physical-like representation, it can certainly be a different one than that which was implemented, as long as there exists a mapping between them. $LTM_{\perp}$ is thus merely a wrapper, or an abstraction of the actual data structure being used. Modifications on instances of $LTM_{\perp}$ are mapped over to changes in the physical dimension, and vice versa. This can be done by having the actually implemented data structure implement an interface as if it were conforming to $LTM_{\perp}$. This requires a mapper between $LTM_{\perp}$ and the physical metamodel, which is similar to physical mappers [295]. Now, however, the mapping is only defined for a single metamodel, instead of for each metametamodel individually, greatly relieving users. This is the mapping shown in Figure 5.18.

Decoupling the implementation of algorithms from the actual internal data structure makes it possible to perform drastic changes internally (*e.g.*, switching between database technologies), without any change whatsoever to the explicit models of model management operations, nor to $LTM_{\perp}$ or $conformance_{\perp}$. Related to this, different tools can implement exactly the same algorithms, which were explicitly modelled, even if their implementation language and internal data structure is completely different. They only need to agree on $LTM_{\perp}$ and the corresponding $conformance_{\perp}$, and an explicitly modelled action language to go along with it. All other implementation choices become truely that: choices made in the implementation that don't affect functionality at all.

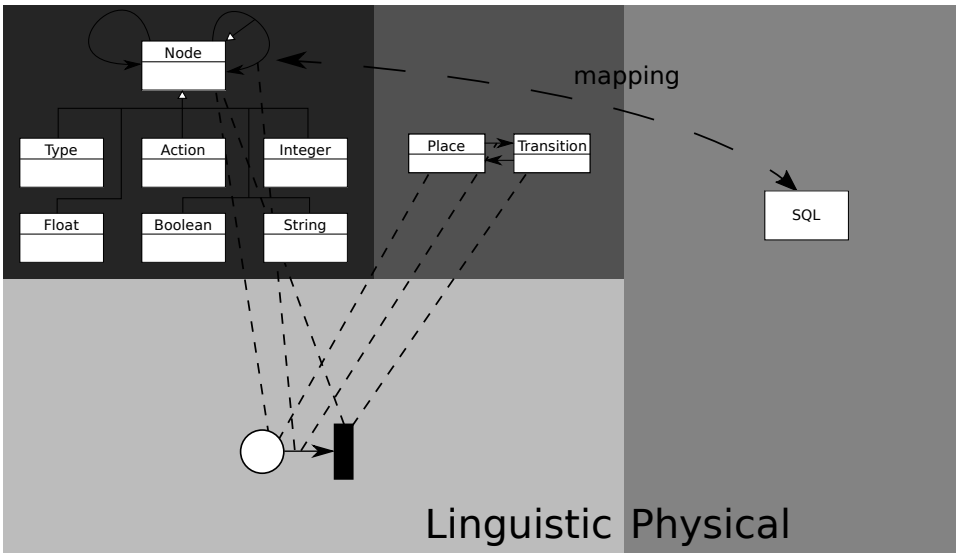


Figure 5.18: Changing the physical metamodel with something else, as long as there is still a mapping to $LTM_{\perp}$. SQL metamodel not expanded due to space constraints.

5.7.3 Evaluation

We now evaluate our approach for two of the claimed advantages. First, we show how a generic model management operation can be executed in the linguistic domain. Second, we show how the physical type model can be altered, while all existing operations are reused as-is.

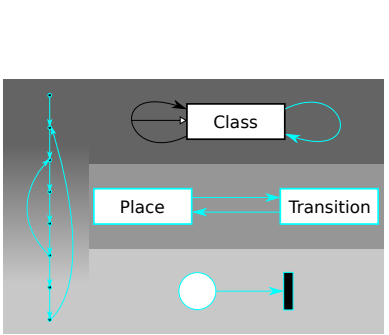
Generic Model Management Operations

As a simple example of our approach, we present here the implementation of a conformance checking algorithm, invoked with a model (a simple Petri Nets model) and metamodel (a simplified Petri Nets metamodel) as its parameters.

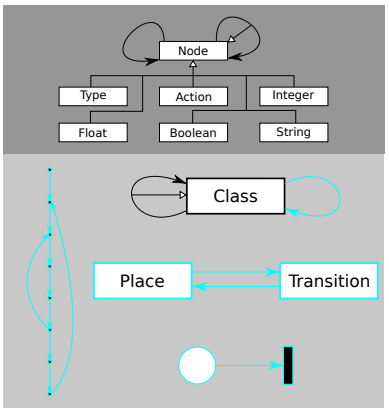
Thanks to our approach, this model management operation can be explicitly modelled, thereby making it portable. The conformance algorithm is defined on the $LTM_{\perp}$ metamodel, and therefore utilises these concepts. Through multi-conformance, both the Petri Nets model and metamodel are deemed to conform to $LTM_{\perp}$, and can thus be passed as parameters.

The conformance algorithm is related to how models are represented internally: all models are subgraphs of a single coherent graph. This format of model representation is itself already level-crossing, as there are edges for navigation. As it contains level-crossing links, it is an invalid model when viewed through an ordinary linguistic typing relation. It is, however, viewable and even modifiable using $conformance_{\perp}$, as the model completely complies to $LTM_{\perp}$.

During the execution of the algorithm, the model is viewed not through the usual conformance relation, but through the $conformance_{\perp}$ relation. As such, the model can be



(a) Using normal conformance.



(b) Using $conformance_{\perp}$.

Figure 5.19: Two different ontological views on the same model. The elements accessed by the algorithm are shown in light blue. Only $conformance_{\perp}$ complies with strict metamodelling.

modified as if it were merely a graph, without any additional semantics or imposed restrictions. Apart from just allowing any kind of structural change, inconsistencies in the usual conformance relation are also possible: cardinalities, multiplicities, potencies, and so on, can all be invalidated as their semantics is not checked at this level: the $LTM_{\perp}$ has no such constraints. Operations defined by the user, using the normal linguistic conformance relation, will just reinterpret the graph to the usual linguistic dimension, thus again checking all additional constraints such as cardinalities.

We use this code to instantiate a new petri net place, as specified by the petri nets metamodel. The example is visualized in Figure 5.19. Figure 5.19a indicates the problem with the instantiation algorithm: it accesses itself and three different modelling levels: the model level to write out the instantiated model, the metamodel level to read out the allowed attributes and all constraints, and the metametamodel level to know about inheritance links and how to handle them. Accessed elements are highlighted in the figure, indicating that the algorithm requires access (and thus, links) to all these levels. It is therefore impossible to add it at either of these levels: adding it to one level would cause violations for the other levels. By taking the $conformance_{\perp}$ view, the modelling hierarchy changes from Figure 5.19a to Figure 5.19b, in which there are no level-crossing links anymore. In Figure 5.19b, all access are again highlighted, but are now within the same level in the modelling hierarchy. There is therefore no longer any violation of strict metamodelling.

The complete procedure is shown in Figure 5.20: first the $conformance_{\perp}$ view is taken on the model, where it is shown as a graph instead of a petri net model and metamodel. Second, this graph is traversed and the requested changes are performed. Finally, the modified graph model is again interpreted using the original conformance relation, where users use their own metamodel and corresponding type mapping to interpret the graph.

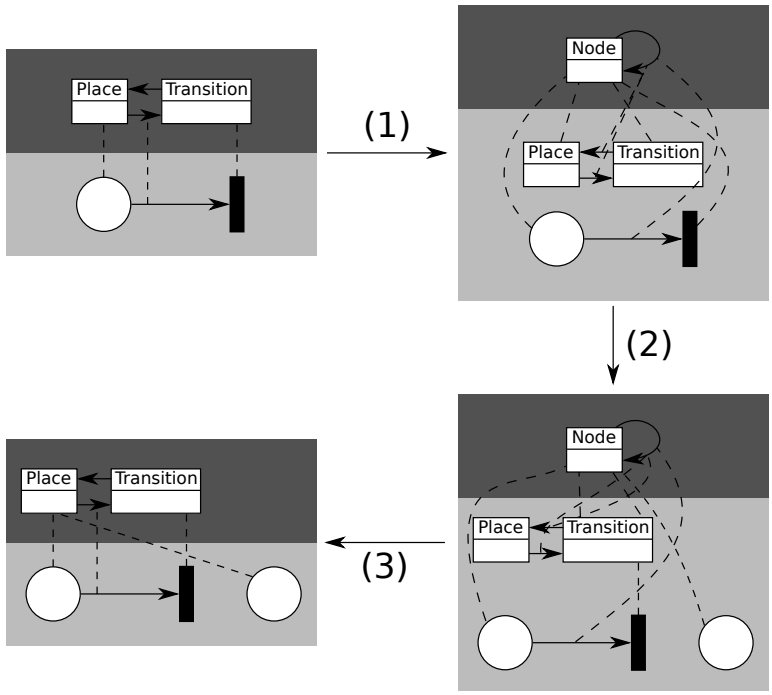


Figure 5.20: Overview of the complete procedure: (1) reinterpret the model as instance of $LTM_{\perp}$, (2) execute the algorithm on the graph representation, (3) reinterpret the model again using the initial metamodel. All steps happen on the background and the user only sees the composite operation.

Altering the Physical Type Model

To illustrate that it is possible to alter the implementation of the MvS, we have implemented multiple MvS implementations in the Modelverse. Apart from a simple MvS implementation whose PTM is identical to the $LTM_{\perp}$, an RDF-based implementation was also created. With the RDF-based implementation, exactly the same $LTM_{\perp}$ is used, and users therefore don't notice the difference, as they never have to access the physical domain anymore. A new mapping between $LTM_{\perp}$ had to be created, however, as of course it is necessary to map operations on $LTM_{\perp}$ to the physical domain, and thus on the actual PTM.

This mapping is done in code, and is therefore not modelled explicitly. In essence, this means that the PTM should have an interface that has an API at the level of the $LTM_{\perp}$.

For RDF, the PTM is a triplestore, saving data in the form of $(\mathit{source}, \mathit{name}, \mathit{destination})$. While this can also be used as a way of storing nodes and edges, it is not necessarily the most appropriate format for doing so. All modifications on this triplestore are implemented using SparQL queries. All code in the Modelverse relies only on the $LTM_{\perp}$, and therefore the switch to a different PTM is not noticeable to any user. A difference is noticeable, however, in terms of performance: some models are naturally better fit to be stored in one format than another.

```
def read_dict(self, node, value):
    if not isinstance(node, rdflib.URIRef):
        return None
    if not self.is_valid_datavalue(value):
        return None

    q = """
        SELECT ?value_node
        WHERE {
            ?main_edge MV:hasSource <%s> ;
                       MV:hasTarget ?value_node .
            ?attr_edge MV:hasSource ?main_edge ;
                       MV:hasTarget ?attr_node .
            ?attr_node MV:hasValue '%s' .
        }
        """ % (node, json.dumps(value))
    result = self.graph.query(q)
    if len(result) == 0:
        return None
    if len(result) != 1:
        raise Exception("Error!")
    return list(result)[0][0]
```

Listing 5.6: SPARQL query for *read_dict*

An example of such a mapping to RDF for the *read_dict* operation is shown in Listing 5.6. This listing presents the Python code for this operation, using the *rdflib* RDF library with SPARQL queries.

5.7.4 Related Work

Three main dimensions of related work exist.

First, our approach builds upon the support for multiple linguistic types. While we have used our approach [314], another possible direction is through by a-posteriori typing [83]. In a-posteriori typing, a model is constructed with a single *constructive* type [27], which cannot be changed. When a model is used in a different context, however, multiple additional types can be added afterwards (*a posteriori*) through the use of concepts [80]. These additional types don't influence the original constructive type, but can make the model applicable for use in other algorithms. Supporting our $conformance_{\perp}$ relation through the use of a-posteriori typing should be similar. The constructive type could simply be part of $LTM_{\perp}$, with all "real" linguistic types specified as a posteriori types. Our approach varies a bit though, since we don't make the constructive type a special kind of type: the $conformance_{\perp}$ is just another relation like any other. The Orthogonal Classification Architecture (OCA) [32] is rather similar to our approach, as it identified the distinction between two conformance relations. But whereas the OCA shifts one of these relations to the implementation level, we merge the physical type model into the linguistic dimension. We therefore still completely comply to the OCA: we have both a linguistic dimension (used for user modelling), and a physical dimension (used during tool building). Parts of our physical dimension are, however, exposed to the linguistic dimension, such that all operations from the physical dimension also become available in the linguistic dimension. With the OCA it is not necessary to support multiple linguistic types for a single model, which is a necessary requirement when shifting more parts to the linguistic dimension.

Second, strict metamodelling has been the subject of several debates, both in favor [28, 32], and against [66, 138]. People against strict metamodelling argue that strict metamodelling makes specific models impossible, as we have also shown in this paper. Their solutions, however, often completely throw away all notions of strict metamodelling. And while we agree that strict metamodelling can be overly restrictive, it certainly has its advantages in protecting ordinary users and simplifying algorithms. So in contrast to tools like XMF-Mosaic [66], who completely flatten the modelling hierarchy, we still enforce strict metamodelling, though users can switch to the "unrestricted mode" by taking on a different linguistic type model. Since the unrestricted mode is at a much lower level of abstraction than the usual linguistic metamodels, users will now have more powerful tools at their disposal, and are able to circumvent strict metamodelling in a controlled way.

Third, many tools rely explicitly on the implementation level. For example, MMINT [100], MetaDepth [79], DISTIL [190], AToM^3 [86], and AToMPM [273] all explicitly allow users to inject code, for example as parts of models, or to extend the capabilities of the tool. Since this code is dependent on both the application interface (API), the implementation language, and the internal data structures, the code is not portable at all. For example, megamodel management [246] is often implemented purely at the implementation level instead of explicitly modelled. And while there is some work on making generic model management possible [237, 325], these approaches often remain specific to the problem under study.

5.7.5 Dynamic PTM Optimization using Activity Models

By making the PTM changeable, possibly at runtime, it becomes possible to optimize the PTM based on the models that we are modifying. For this, we link to the domain of activity models, which we have investigated in the domain of DEVS simulation [303, 307, 315]. We briefly present this work here, and then elaborate on how this can be applied in the Modelverse as future work.

Activity Models in DEVS

DEVS is a popular formalism to model system behaviour using a "discrete-event" abstraction. It frequently serves as a simulation "assembly language" to which models in other formalisms are translated, either giving meaning to new (domain-specific) languages, or reproducing semantics of existing languages [319]. Models in different formalisms can hence be meaningfully combined by mapping them onto DEVS.

But while domain-specific simulators can make internal optimizations based on many assumptions, this is not possible for a general purpose formalism such as DEVS. The mapping to DEVS, therefore implies reduced performance, as information is lost in the translation to a semantically equivalent model, executed in a domain-agnostic simulator. Through the use of activity models, we attempt to encode this domain information and make it available within the DEVS simulator [303, 307, 315].

Resource Usage-based Optimization We illustrate the use of activity models through a well-known optimization: load balancing. When a model is being simulated over multiple processes, it can be detected that some processes have more work to do than others. In

reaction to this, the model allocation can be altered, thereby balancing the load of the simulation. It is important to note that this new allocation is based on the measured load during the execution, and therefore relies on the past. As such, load balancing always optimizes the future for values measured in the past. It can therefore be the case that load balancing lags behind and is actually counter-productive, should it "optimize" and actually make the allocation worse for the future.

For example, Figure 5.21 shows a snapshot of an air traffic simulation, with each yellow dot indicating a plane. The current core allocation is also shown, putting America and Europe/Africa on the same core, and Asia on another. Somewhat later, the load changes to the situation shown in Figure 5.22, where load balancing detects that the load is no longer balanced. As such, load balancing changes the allocation as shown in Figure 5.23, where now America is put on its own core, with Europe/Africa and Asia sharing the other one. While the load is now again balanced, it is quickly disturbed again. Depending on how fast simulation progresses, load balancing lags behind, causing load balancing to actually be counter-productive in this case: the allocation shown in Figure 5.23 is soon outdated when it becomes night in America.

Encoding Domain Knowledge Load balancing based its knowledge on the measured values during the running simulation. Much of this knowledge, however, is inherent to the domain and is known a-priori by the modellers, but not known by the simulator. For example, with air traffic, it is well known that there are not as many flights at night than during daytime. In our case, load balancing has similar results, but takes some time to measure the current load, and only afterwards does it balance based on this. As the past is used to optimize for the future, this might not be a correct prediction: as daytime shifts throughout the simulation, so does the load. When the load is analyzed for the situation shown in Figure 5.22, Figure 5.23 is a logical result based on the currently observed load. In the near future, however, the load will continue to shift, maybe making the load balancing overhead higher than the expected gains.

By encoding some domain knowledge, for example the mere fact that night implies not a lot of air traffic and thus not a lot of load, it becomes much easier to determine the allocation. Indeed, instead of measuring the execution over some time period and making assumptions based on that, we can make statements based on the current situation, without taking into account the past. While this does not decrease the load balancing overhead too much, allocations can be made before the load changes. For example, if it just became daytime on a continent, load balancing mostly ignores this continent, as not much load has been measured here in the past. With domain-knowledge, however, we can assume that if it is morning, there is already some load. The duration in which the allocation is optimal therefore becomes longer, making the allocation more worthwhile.

Resource Usage Predictions A logical extension to this is to predict what will happen in the near future. Load balancing actually tries to do this by looking into the past and assuming that the future will be highly similar to the past. This is, however, not always the case. Even looking at the current situation might not be sufficient to come up with a good allocation, depending on how fast the load migrates throughout the simulation. Another factor to take into account is how much performance gain we project to achieve through this optimization. For example, the load might shift so fast that the overhead of performing

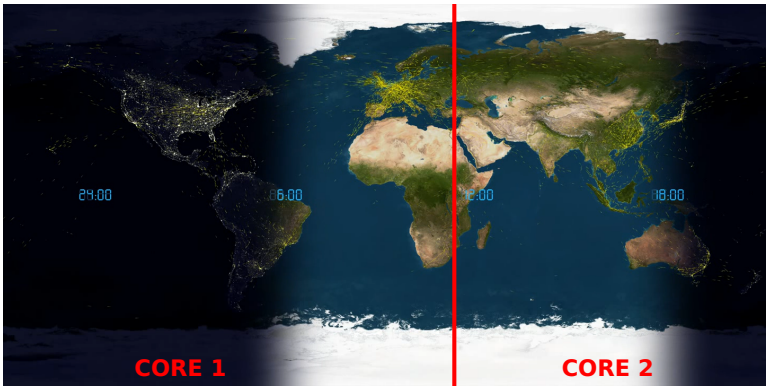


Figure 5.21: Snapshot of air traffic simulation load distribution at start of simulation.

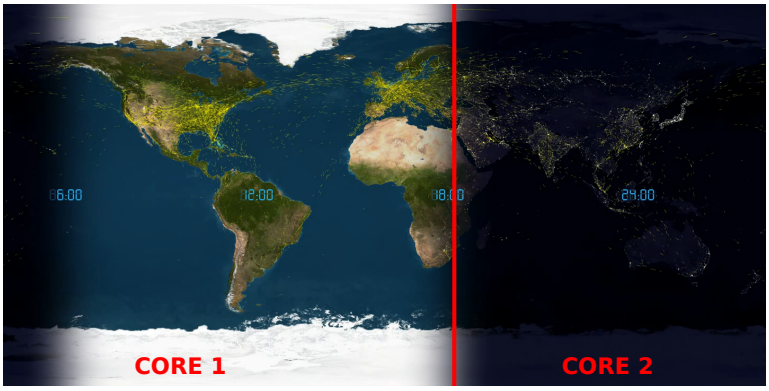


Figure 5.22: Snapshot of air traffic simulation load distribution during simulation.

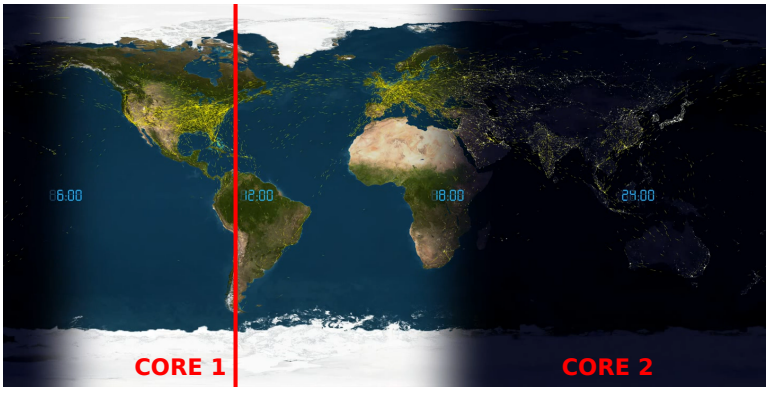


Figure 5.23: Snapshot of air traffic simulation load distribution after load balancing.

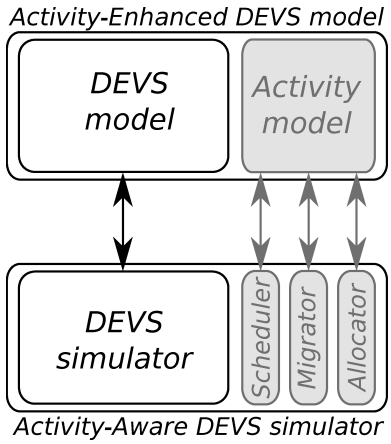


Figure 5.24: Activity as an optional extension to both the model and simulator.

the migration is not worth it compared to the performance improvement that we expect. By making such predictions, it actually becomes possible to optimize for the future, instead of for the past.

Such predictions can happen through the use of an activity model. This activity model is a model similar to the model being simulated, but probably in a different formalism and at a higher level of abstraction. When predictions about the load distribution in the future are made, this abstract model can be simulated to determine the likely load distribution. Due to the level of abstraction, this projected load distribution is likely not correct, and might indeed be completely incorrect. It is therefore important to guarantee that, independent of the results given by the activity model, simulation will always give correct results. This activity model is only an extension to the model being simulated: results should be correct whether or not the activity model is used. Simulators that can make use of this additional information are called "activity-aware". Both the activity model and activity-awareness in the simulator are optional: only if both are provided, is the activity information actually used. This allows for portability of the model with activity-unaware tools. This is shown in Figure 5.24.

Results Results for a simplified city-layout model are shown in Figure 5.26 for a model structured as in Figure 5.25. This city-layout model simulates the behaviour of car traffic in rush hour, with several "residential areas" (traffic source) and "commercial areas" (traffic destination). When going from the residential to the commercial areas, cars have to pass through several districts, which can be the atomic units for load balancing. The results include four different configurations. The first is "no activity tracking", which does no balancing of the load at all: in the beginning of the simulation each process is assigned an equal number of districts. The second is "activity tracking", which is equivalent to load balancing: execution times in the past are used to optimize for the future. The third is "custom activity tracking", which looks at the current configuration (e.g., number of cars in a district) and optimizes for that. The fourth is "custom activity prediction", which looks at the current configuration to make predictions about the future. Results indicate that the use of load balancing indeed increases performance in some situation (where the

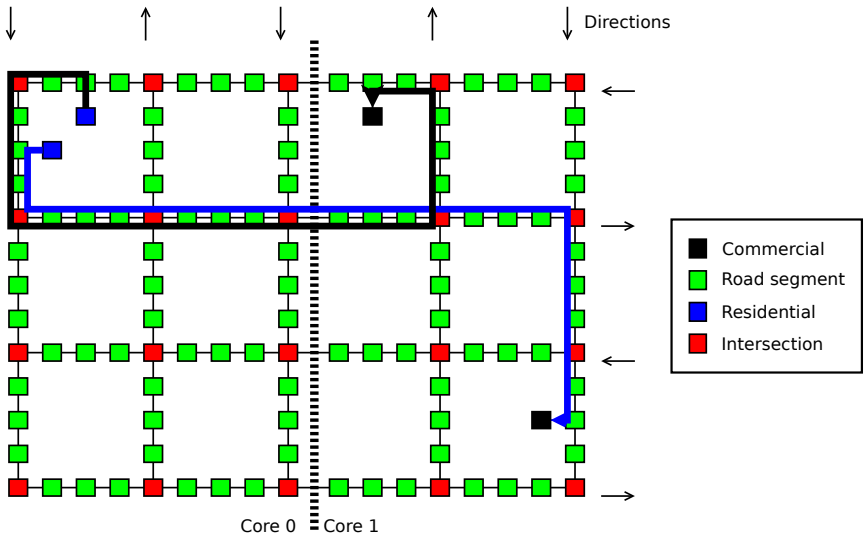


Figure 5.25: Simplified city lay-out model.

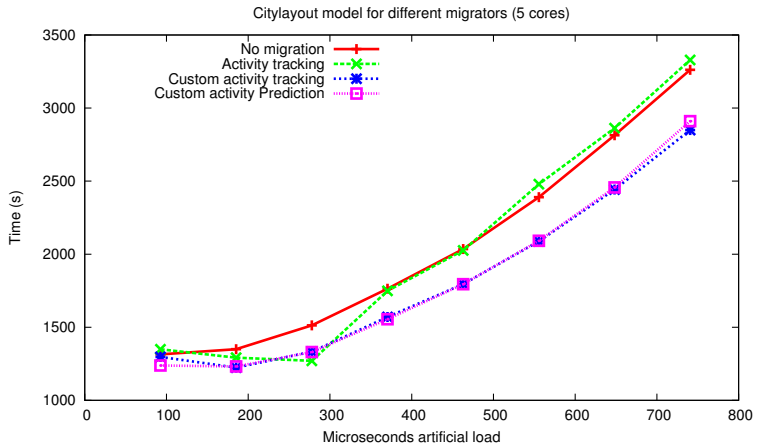


Figure 5.26: Execution time results for the city layout model.

history is an accurate representation of the future), but actually decreases performance in other scenarios. This is due to the past being non-representative for the future, causing changes to be ineffective, while still inducing the load balancing overhead. Using custom activity, which has domain knowledge encoded in it, results are always faster than no activity tracking, as the past is ignored and only the current situation is considered. No significant benefits are seen from prediction in this case, as the present is sufficient to determine the future.

Data Structure Optimization Up to now we have only considered the use of activity models for load balancing, to optimize the allocation of models during runtime. Other

	Average case	Worst case
List	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Heap	$\mathcal{O}(k \cdot log(n))$	$\mathcal{O}(n \cdot log(n))$

Table 5.1: Complexity of different scheduler types. k is the number of imminent models and n is the total number of models in the simulation.

applications are possible as well, such as for chosing the optimal scheduling data structure. Indeed, depending on the domain, the optimal scheduling data structure varies. We consider two types of scheduling data structure: a heap (with the usual operations) and a list (which is iterated over to find the lowest entry). For the heap, the usual heap push and heap pop operations are required to manage the data structure, both having complexity $\mathcal{O}(log(n))$ for a single element. For the list, the list is always completely iterated to find the earliest element, having a complexity of $\mathcal{O}(n)$ for all elements. These complexities are shown in Table 5.1. We consider two parameters: k being the number of imminent models and n the total number of models in the simulation. An imminent model is a model that is about to transition at the next point in simulated time. There can be multiple such models, if multiple models execute their transition at the same point in simulated time. We notice that if several models are imminent simultaneously, the heap might not be the most efficient data structure, as its complexity goes to $\mathcal{O}(n \cdot log(n))$, compared to the $\mathcal{O}(n)$ of the list.

Which data structure is ideal depends on the pattern exhibited by the model, which is often tightly related to the domain. For example, firespread simulation [206] is often done using discrete-time simulation, for which the list-based approach is ideal. As such, we have defined a "polymorphic" scheduling data structure, which adapts its internal representation depending on the access patterns that it notices. When many imminent models are detected, the heap is restructured to a list, and when few imminent models are detected, the list is restructured to a heap. Performance results are shown in Figure 5.27. All results are normalized to the polymorphic data structure, showing that the polymorphic data structure is always close to the fastest structure, independent of the percentage of "collisions" (i.e., imminent models). While in most cases, the polymorphic is only slightly slower than the fastest scheduler, it is always significantly faster than the slowest scheduler.

Another use case is when the ideal data structure changes throughout the simulation, for example because there are different phases in the simulation: some with many imminent models, and some with few imminent models. Performance results are shown in Figure 5.28. The same simulation is ran with the three scheduling data structures, showing that the polymorphic scheduler is again always closest to the fastest data structure. Despite the additional overhead of monitoring access patterns, the polymorphic scheduler is the fastest scheduler overall, as it always picks the best option. It is thus possible that the overhead is mitigated.

Protocol Optimization Another optimization that is possible, is chosing the correct synchronization protocol for parallel simulation. The synchronization protocol is responsible for how the different processes, which are at different points in simulated time, are to synchronize their data in a consistent way. If no synchronization protocol were to be used, simulation results would be incorrect.

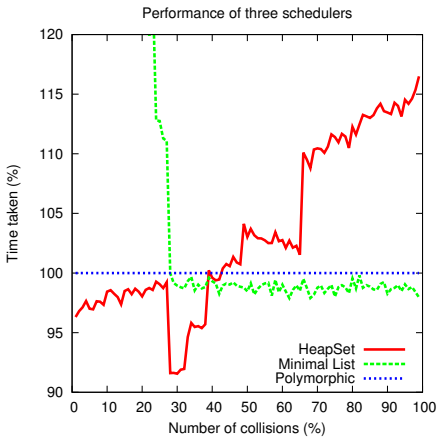


Figure 5.27: Results of using the polymorphic scheduler data structure, normalized for the polymorphic scheduler.

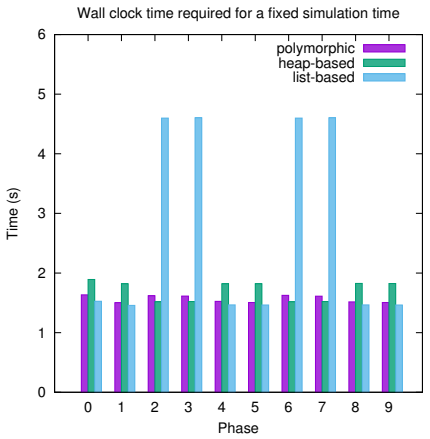


Figure 5.28: Results of using the polymorphic scheduler data structure with different phases.

Roughly speaking, there are two categories of synchronization protocol: conservative and optimistic [115]. Conservative synchronization allows each process to progress up to some point in simulated time, calculated based on guarantees given by the processes. When the predefined time is reached, the process pauses until it receives a new guarantee. This protocol is highly efficient when strong guarantees can be made, but can easily grind to a halt if no such guarantee can be made. Optimistic synchronization allows each process to progress as fast as possible, but accounts for so-called *straggler events*. A straggler event is an event that should have occured in the past of the process that receives it. To still process the straggler event, simulation has to be rolled back to that point in time, when the event can be processed as usual. This protocol is highly efficient when communication is only sporadic and if the load is nicely distributed, but can easily grind to a halt if many rollbacks are required.

There are thus two synchronization protocols with widely different performance [117]. Depending on the model being simulated, the synchronization protocol has to be chosen. This is problematic, however, as modellers can often not foresee the exact behaviour of their model. Even if they can, the type of synchronization protocol is an implementation detail that is far from the problem domain, and as such modellers are either not aware of it, or don't fully understand the implications.

Current DEVS simulators have often only implemented a single synchronization protocol, thereby rendering the choice unnecessary. This is not the soluation, however, as a model might then simply become unsuited for the simulator in terms of performance. Depending on the configuration of the model, the ideal type of synchronization protocol might even change.

One such example benchmark is shown in Figure 5.29, where the percentage of priority events, which depends on a parameter, influences the ideal synchronization protocol. The model is the usual PHOLD performance benchmark [116], but extended with high-priority events. High-priority events are events that are sent nearly instantaneously. Such events

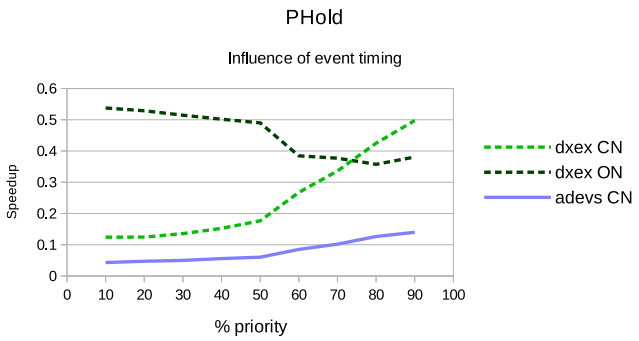


Figure 5.29: Performance of conservative and optimistic synchronization for the modified PHold model.

decrease the guarantees that can be made, as the chance of such a high-priority event must be taken into account. When nearly no such events are sent, however, we see that conservative (CN) synchronization suffers: high-priority events are taken into account, but almost never occur. The full potential is then only achieved with optimistic (ON) synchronization, which does not rely on such guarantees. When almost all events are high-priority events, the guarantees actually become representative of what is to be expected, and conservative synchronization becomes better. Optimistic synchronization now suffers, as these high-priority events cause rollbacks.

It is thus important that a single DEVS simulator implements both types of synchronization protocol, to adapt to the model, instead of vice versa. Our tool Dxex [59] was extended with multiple synchronization protocols, which can be switched at runtime [60]. By making this choice at runtime, based on the number of rollbacks or the guarantees that are received, the simulator can switch synchronization protocol, thereby increasing performance. As the model's behaviour and resource usage is used to optimize the running simulation, this is considered to be an activity-based optimization as well.

Relation to Machine Learning This approach bears some similarities to machine learning, they are complementary. Machine learning can be used to find complex patterns, though does not depend on domain knowledge directly. It might therefore be possible that some patterns are learned that make no sense in general. Nonetheless, machine learning can automatically find complex activity models that would be too complex for domain experts to code. The use of activity models explicitly relies on domain experts providing additional information, which domain experts believe to be correct. In the limit, it could be possible to use machine learning to come up with activity models, which are further augmented by the domain expert, or vice versa.

Activity Models in the Modelverse

The previously mentioned results were obtained by applying activity in the context of DEVS simulation. Going back to the Modelverse, we believe that similar results can be achieved in the context of the Modelverse. Indeed, the Modelverse also makes several decisions that

could potentially be optimized, such as the choice for the PTM to use. Depending on the chosen PTM, the performance can hugely vary: if a matrix-like model is stored using a PTM for matrices, all matrix operations become native operations. If, on the other hand, this same matrix-like model is stored using a generic graph-like PTM, then these same operations can become complex graph operations.

Thanks to the explicitly modelled PTM, which is actually the $LTM_{\perp}$, it became possible to alter the PTM without any effect on the operations and models. As such, an activity model, related to the models in the Modelverse, can be used to hint the Modelverse to use a different PTM for this model specifically. This dynamic switching of PTM, and in particular based on hints of an activity model, is future work at the moment. Nonetheless, our previous experience with activity models indicates that this optimization would have significant potential.

Summary

As the PTM no longer has to be static, it becomes possible to optimize the choice of PTM based on execution information. In the domain of DEVS simulation, previous work showed the potential benefit of optimizing the runtime for the model. An example was the optimization of the internal data structure depending on access patterns that were noticed and predicted for the future. For the Modelverse, it would similarly be possible to optimize the PTM depending on the type of model we are manipulating. For example, a model representing a matrix could be stored in a native matrix storage format, instead of as a graph structure. We project this to have a significant benefit in terms of performance and memory consumption.

5.7.6 Link to Requirements

Explicitly modelling the physical type model, and allowing users linguistic access to it, has an influence on **Requirement 1 (Language engineering)**, **Requirement 2 (Activities)** and **Requirement 10 (Portability)**.

Requirement 1 (Language engineering) is influenced, as the new $LTM_{\perp}$ formalism can be considered as a "new" language in which model management operations can be modelled. The activity-based optimizations could certainly be of use when domain-specific languages are modelled that are not necessarily best represented as a graph (e.g., matrices).

Requirement 2 (Activities) is influenced, as activities can now also be used to define the low-level operations on the model, which normally would have to be implemented on the physical representation of the model directly. This therefore extends the applicability of activities: both model transformations and procedural code.

Requirement 10 (Portability) is influenced, as the physical type model now becomes independent of the operations that ordinarily operate at that level (e.g., model management). As the physical type model is now split from the models at which all low-level operations happen, the physical type model can easily be altered without any influence whatsoever. This allows divergent implementations of the physical level.

Summary

The Physical Type Model (PTM) stores the physical representation of the model, and is often accessed to implement generic functions (e.g., conformance checking). For pragmatic reasons, many model management operations are implemented directly in the language of the implementation platform (e.g., Java), directly modifying the internal model representation (e.g., XMI). Crucial model management operations thus become dependent on implementation details and are therefore non-portable. By explicitly modelling the PTM in the linguistic dimension (the $LTM_{\perp}$), generic functions can be modelled explicitly in the linguistic dimension. Algorithms can then be modelled explicitly as operating on $LTM_{\perp}$, which offers the same benefits as coding them for the PTM, while also having the benefits of explicitly modelling it. This ensures that model management operations are not grafted on the tool, but on another model ($LTM_{\perp}$), thereby offering flexibility. When combined with the prospect of activity models, this technique can also be used to automatically switch the PTM depending on which model is being manipulated, for example for increased performance.

5.8 Service Orchestration

As the Modelverse supports a procedural action language, most algorithms can be directly implemented in the Modelverse. Reimplementing all algorithms in the Modelverse, however, incurs several problems. First, the algorithm might be much slower in the Modelverse than in some other programming language or some other tool. For example, many optimizations exist to constructing a reachability graph, which sometimes utilize details of the language (e.g., data structures, library support). Second, the algorithm might be too difficult to port in a reasonable amount of time. For example, simulation algorithms of specialized tools are often the result of multiple man-years of development, making it infeasible to replicate the effort. Third, this procedural action language might not be the most appropriate formalism to describe the problem and its solution. For example, logical problems are ideally solved using logical programming languages, such as ProLog. Therefore, most algorithms should not be implemented in the Modelverse directly, even though it would be possible.

Nonetheless, the Modelverse must support such operations, as operating on models is one of the requirements of the Modelverse. Such external algorithms can be invoked through the use of services. A service is an external tool that offers a specific operation, which can be invoked from the Modelverse. This external tool does not have to be a (meta-)modelling tool, though that is possible. For example, a service could just as well print out a text file, therefore taking in a model in the *Text* modelling language. This makes the Modelverse a "service orchestrator": existing (proprietary) tools do the heavy lifting, but the Modelverse orchestrates when to execute them, and on what models to execute them.

An additional dimension to take into account, is that the user can be considered as a service. Indeed, a service is merely defined as an entity external to the Modelverse, which the Modelverse can contact to perform a specific action. The Modelverse is oblivious to what this service is, and it might very well be a human performing some specific (manual) job.

5.8.1 Motivation

Due to the increasing complexity of today's systems, not only the number of involved languages, but also the number of involved tools is increasing. By explicitly modelling the process, we have define when an external tool is to be invoked (as an activity), but not how it is to be invoked. Given that these external tools are often complex, and designed as stand-alone entities, interaction with such tools is non-trivial. To orchestrate such variety of tools, it is therefore important to define how to interact with them.

Orchestration requires a detailed specification of the interaction protocol with external services. In manual activities, user input is required, often through a (visual) modelling and simulation tool (for example, to create a model). In automated activities, a service (or multiple services) might be invoked and communicated with in an automated way (for example, to run a simulation). Such interaction protocols exhibit timed, reactive, and concurrent behaviour, making their formal analysis paramount in industrial-scale engineering processes. The analysis of the interaction protocols can improve its overall process with regards to transit time, scheduling, resource utilization, and overall model consistency. These interactions are, however, typically specified in scripts or program code, which interface with the API of the tools providing the services. Such an encoding of the interaction protocols inhibits their formal analysis.

Activities can be hardcoded, but code is arguably not the optimal formalism to describe an activity. While activities can be limited to executing some local computation, it frequently requires external tool interaction. Such external tools can be anything, for example a (highly-optimized) simulator, or a modelling tool. In such cases, hardcoding the potentially complex interaction protocol is far from ideal. Indeed, the behaviour of protocols exhibits timing (*e.g.*, network timeouts, delays), reactivity (*e.g.*, responding to an incoming message), and concurrency (*e.g.*, orchestrating multiple tools concurrently).

We propose to explicitly model the external service interaction protocols in the activities of engineering processes using SCCD [298], a variant of Statecharts [133]. SCCD is appropriate for modelling timed, reactive, autonomous, and dynamic-structure behaviour, as it has native constructs available for it. Indeed, concurrency is supported by orthogonal components and dynamic structure, reactivity is supported by event-based transitions, and timeouts are supported by *after* events. This facilitates the implementation of the interaction protocols, and enables future analysis of the service orchestration.

Motivating Example

Our motivating example is in the domain of optimization. A set of configuration parameters go into the activity and the ideal configuration, including its cost, come out. For example, the number of traffic signals in a fixed-length railway system. The system consists of sequences of railway segments, each guarded by a single traffic signal. For safety reasons, only one train is allowed on each railway segment, despite the segment being longer. Adding more traffic signals increases the throughput of the system, though also increases the cost of maintenance. The ideal number of traffic signals is therefore dependent on the characteristics of the system (*e.g.*, train inter-arrival time, acceleration, total length of the track). An example process for this is shown in Figure 5.30, which includes the optimization as yet another activity.

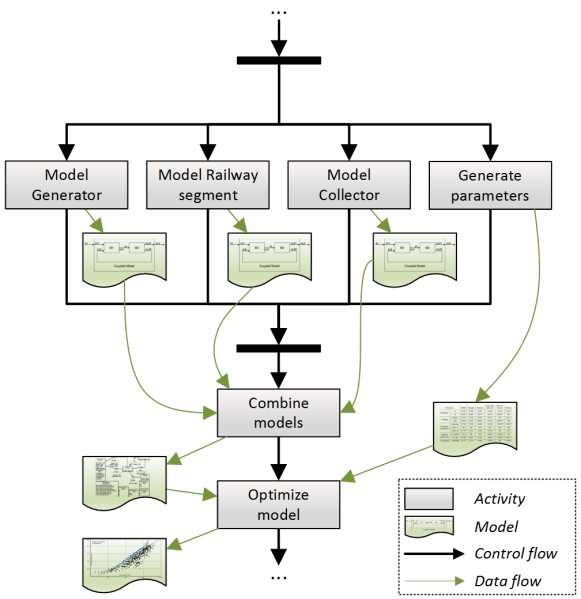


Figure 5.30: Process model of the example.

We consider the optimization problem in the context of the DEVS formalism [338], which is often used to assess the performance of queueing networks. The incoming model is simulated for a fixed set of parameters, with some other parameters being varied. All simulation results are collected, the cost function is evaluated for all of them, and the configuration with the minimal cost is returned. In essence, the same simulation is ran with slightly different parameters. This is, however, embarassingly parallel: each simulation run is independent of every other simulation run. Therefore, we desire to run some simulations in parallel. Doing this the usual way (*i.e.*, with code) is non-trivial: concurrency requires threads (which is problematic [176]), reactivity requires the use of a main loop (possibly with polling), and timeouts require interruptable sleep calls.

5.8.2 Model

Activities are the atomic actions being executed throughout the enactment of the process. Up to now, we were agnostic of what is the content of the activity, as we merely require it to be executable. Most often, it is hardcoded in some programming language or provided as an executable binary. When control is passed to a specific activity, the activity executes.

In our running example, we see that these features of SCCD are all required. Concurrency is required to spawn several instances of the simulator concurrently, and the number is only known at runtime, as it depends on the number of possible configurations. Reactivity is required to handle the results of these individual simulators, which should be aggregated. Timeouts are required to handle network timeouts, as the external service is communicated to over the network, and potential infinite simulations. In Figure 5.31, we show how the example activity is modelled with SCCD.

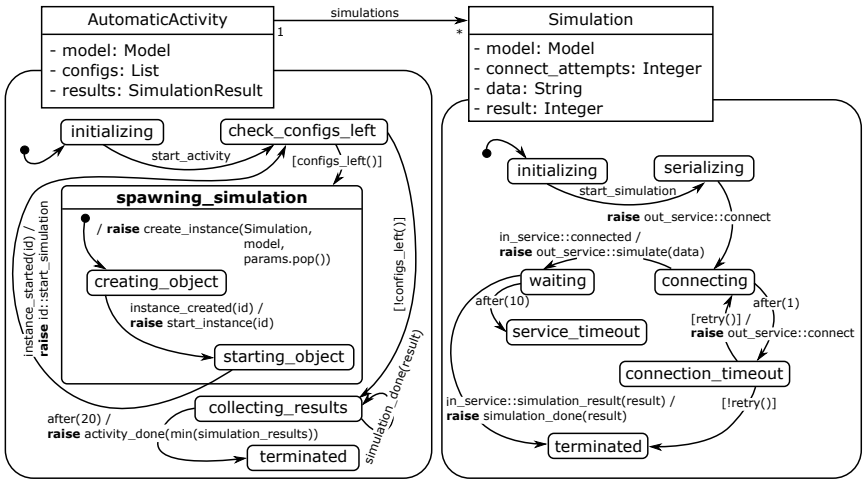


Figure 5.31: Automatic activity: protocol implemented to communicate with an external service.

5.8.3 Evaluation

We implemented this example in the Modelverse [304, 316], our prototype Multi-Paradigm Modelling tool. Simulations were performed using the PythonPDEVS [308] simulator as an external service. To transfer the model from the Modelverse to PythonPDEVS, the model was written out to a file format supported by PythonPDEVS, of which the content was transferred to PythonPDEVS. After the simulation, the simulated cost was forwarded back to the service orchestrator.

Thanks to SCCD, we can spawn an arbitrary number of *"Simulation"* objects, by sending out an event to the object manager, thereby allowing for dynamic structure (implementing *concurrent*). After a simulation is spawned dynamically, for each configuration to evaluate, we wait for results to come in, encoded in events (implementing *reactive behaviour*). Each of the spawned simulations serializes the model, and sends it to the actual external simulator, after which the simulator is started externally. If no response is received from the simulator during initialization before a timeout occurs, we retry the connection (implementing *timing*). If the simulation was started successfully, but no result comes in before a timeout occurs, we determine that the simulation has crashed, is stuck in an infinite loop, or ran out of memory. Independent of the reason, we determine that the simulation result is not the optimum, and subsequently ignore the simulation run. When all simulation results are in, or we have waited sufficiently long, we return the optimal parameter that we found.

5.8.4 Related Work

The orchestration of different services is tightly interwoven with process and workflow modelling, which is an extensively researched domain. Process modelling languages are primarily geared towards modelling concurrency and synchronisation [286]. Pertinent examples include languages based on the *Business Process Modelling Notation* [269], *Petri Nets* [284] and *UML Activity Diagrams* [45]. We focus on and discuss the most relevant and well-known approaches in terms of the intentions of this paper next.

The *Business Process Modelling Notation* (BPMN) [260] is a widely used standard in process modelling. BPMN is used in a wide range of areas, to model processes in non-IT, as well as IT-intensive organisations. Its main goal is to provide an understandable notation for all stakeholders. The focus is more on the conceptual modelling of processes, and less on orchestration. In version 2.0, the standard has been extended with support for orchestration, albeit on a non-technical level.

jBPM [73] is an open-source, Java-based framework that supports execution of BPMN 2.0 conform processes. The framework also provides enhanced integration features with external services in the form of managed Java program snippets. In addition, the process engine is tightly integrated with a collaboration and management service (Guvnor), a standardized human-task interface (WS-HT), a rule engine (Drools) and a complex event processing engine (Drools Fusion).

The *Business Process Execution Language* (BPEL) [330] is a standardised language for specifying activities by means of web services. The standard specifies a BPEL process as XML code, though graphical notations exist, often based on BPMN. Service interaction can be executable or left abstract. Analysis tools for BPEL have been developed, for example by formalising BPEL models in terms of *Petri Nets* as done by [215] and [334]. [167] use a symbolic analysis model checker. [114] and [111] analyse the communication between BPEL processes by employing automata. Nevertheless, BPEL is exclusively used for web services defined using WSDL.

Yet Another Workflow Language (YAWL) [285] attempts to combine the functionality of BPMN (business-mindedness) and BPEL (executability). In contrast to other approaches, YAWL was designed with formal semantics in mind, and is defined as a mapping to *Petri Nets*. Execution particularly aims to provide insight in data and resources. There is, however, no particular focus on the integration and orchestration of tools.

Orc [160] is a formal textual language for the orchestration of service invocation in concurrent processes. It aims to manage timeouts, task priorities, and failure of services and communication. Orc is based on trace semantics, which is used to determine whether two Orc programs are interchangeable. The integration of tools can be achieved by defining sites, which represent units of computation. There is no support, however, for modelling modal behaviour, and the textual notation does not scale to large processes.

Open Services for Life-cycle Collaboration (OSLC) [213] is the de facto standard in tool integration. It is a specification for the management of software lifecycle models and data, which are represented as resources. The specification is intended to be used for integration of services and data, and does not include process modelling.

The *Statecharts* formalism [133] has first-class notions of concurrency, hierarchy, time and communication. It can therefore be viewed as a suitable formalism for integration and orchestration. Because *Statecharts* is state-based, and does not include *fork* and *join* constructs, it is less suitable for process modelling. *Statecharts* has been combined with *Class Diagrams* in SCCD [298], to provide structural object-oriented language constructs (*i.e.*, objects with behaviour).

Story diagrams [327] are a formal behavioral specification language with workflow semantics. Similarly to UML Activity Diagrams, they describe control and data flow across the workflow, but with the added support for specifying executable actions. Just like

the FTG+PM formalism, story diagrams rely on typed attributed graph transformations, but with a very simplistic type model. As a result, event though story diagrams provide added behavioral specification semantics, the formalism still is not as versatile as the FTG+PM.

Biehl *et al.* [46] proposed Tool Integration Language (TIL): a DSML to describe a service-oriented tool chain, which could be used for communication, design and automated generation. Additionally, support for analysis and evoluation of the tool chain is planned.

A summary of all approaches and their suitability for our purpose is presented in Table 5.2. We have investigated whether the approach is intended to be used to specify processes (**process**), whether it aims at integration of services/tools (**integration**), whether it supports execution or enactment (**executability**), whether it provides means for formal analysis (**analyzability**), and whether its notation is appropriate for the tasks it is intended for (**usability**).

Approach	Process	Integration	Executability	Analysability	Usability
Petri Nets	●	○	●	●	◐
Activity Diagrams	●	○	●	●	●
BPMN2.0	●	○	●	●	●
jBPM	●	●	●	○	●
BPEL	●	◐	●	●	◐
YAWL	●	○	●	●	●
Orc	●	◐	●	●	○
OSGi	○	◐	●	○	●
OSLC	○	●	●	○	●
FTG+PM	●	○	●	●	●
SCCD	○	●	●	●	●
Story diagrams	○	●	●	◐	◐

Table 5.2: Summary of related work. (● - Supports, ◐ - Partially supports, ○ - Does not support)

5.8.5 Link to Requirements

Explicitly modelling service orchestration influences **Requirement 2 (Activities)**, **Requirement 4 (Multi-User)**, **Requirement 5 (Multi-Service)**, and **Requirement 10 (Portability)**.

Requirement 2 (Activities) is influenced, as activities can now communicate with external services, while still retaining many of the advantages of being explicitly modelled.

Requirement 4 (Multi-User) is influenced, as multiple users, in the form of manual activities, can be orchestrated in this way to enhance the multi-user experience. Indeed, if a user is seen as an external service, it becomes possible to orchestrate not only tools, but also users performing specific jobs.

Requirement 5 (Multi-Service) is influenced, as this offers an explicitly modelled way of interacting with existing external services, possibly several simultaneously. The use of SCCD to model the protocol, and the underlying communication which happens through XML/HTTPRequests as usual, also does not restrict the set of services that can be connected.

Requirement 10 (Portability) is influenced, as reuse of existing operations, such as those of existing programs, can be reused this way, without requiring a new implementation. As we exclusively communicate through sockets, our approach is independent of the programming language in which the service-side wrapper is defined, as well as the implementation language of the external service.

Summary

In the context of MPM, service orchestration is essential for the combination of multiple external tools. Nonetheless, current approaches do not sufficiently address the challenges posed by tool interaction: timed, reactive, and concurrent behaviour. Effectively, the communication with an external tool is dictated by a protocol, for which we have previously found SCCD to be the most appropriate. We proposed an approach for handling this problem by modelling the activity using SCCD (a *Statecharts* variant), which has native notions of timing, reactivity, concurrency, and dynamic structure. Thanks to the use of SCCD, we achieved an intuitive way of interacting with an arbitrary number of external tools, independent of the implementation language or location of these tools. Additionally, the use of SCCD allows for analyzability.

5.9 FTG+PM Enactment

Apart from process modelling, which can aid as documentation, the Modelverse should also support its enactment. The advantages of full support for the FTG+PM have already been described in the literature [185, 186, 204]. To recapitulate, enactment of the FTG+PM allows complex processes to be not only useful for documentation, but also for execution. When a complex workflow is described, together with several languages, it can automatically be enacted by executing the linked activities in the specified order. Frequently, several activities are to be executed concurrently, potentially by different users (e.g., one by the plant engineer and the other by the control engineer). By doing this enactment automatically, it is ensured that the process is followed, and the modellers are maximally assisted (e.g., the correct languages and interfaces are loaded in advance).

5.9.1 Motivation

Process models aim at depicting how the various domain-specific models are used during development. Models are passed around in the process and are being worked on within the activities of the process. These activities are either manual or automated, and typically make use of various services offered by engineering tools. If modelled in an appropriate formalism, the process can be analysed and subsequently enacted [214]. The enacted process orchestrates the engineering services, thus enabling a higher level of automation in the flow of the modelling work in general.

We provide execution semantics for the overall process, expressed as an FTG+PM model, by mapping the PM part to an SCCD model. This avoids the need to define operational semantics for activity diagrams, which is non-trivial. A naive implementation merely dictates an arbitrary order of the concurrent activities, without actually executing them concurrently. This was originally the case in our prototype tool, since true concurrency is difficult and relies on many platform characteristics. Some simple examples are the choice

between processes or threads, their interleavings, parallelism support of the implementation platform, and how data is shared between activities. This is only a small selection of crucial questions regarding the implementation of process enactment. A significant investment is therefore needed to implement and maintain this infrastructure using traditional (code-based) techniques.

The process model chains the different activities, dictating the order in which they should be executed, possibly concurrently, and on which models. Of specific interest is the fork/join operation, which executes multiple activities concurrently and synchronizes when both have finished. This is ideal for manual activities, involving multiple developers, who can now model concurrently.

The lack of native concurrency in many implementation languages causes problems, as implementing process model enactment requires many workarounds. There are, however, languages that natively support concurrency, such as SCCD. Nonetheless, none of these languages was designed to model workflows, and is therefore unsuited for direct modelling. In summary, we want users to model using activity diagrams as usual, but we transform the modelled process to an SCCD model for execution purposes. This transformation defines denotational semantics for process models, instead of operational semantics (an executor). While other similar languages exist, we favour SCCD, as this allows us to reuse the existing SCCD execution engine and this provides some future opportunities (see next section). Additionally, by mapping to SCCD, there is only one implementation of an executor for timed, reactive, autonomous, dynamic-structure behaviour that must be maintained (the SCCD executor).

5.9.2 Model

We map these activity diagrams to SCCD through the use of explicit transformation models. In our case, the LHS contains activity diagrams elements, such as the *activity* construct, and the RHS copies the activity diagram construct (leaving the activity diagram intact) and creates an equivalent SCCD construct (*i.e.*, an orthogonal component). Defining this model transformation is significantly less work than defining operational semantics from scratch.

A basic mapping to SCCD consists of mapping forked activities to orthogonal components, that each spawn and manage the execution of the activities; joins synchronize the execution by transitioning from the end states of these components. While intuitive, this mapping can run into problems, as an analysis of all concurrent regions would be necessary. For example, consider two parallel forks that interleave, as shown in Figure 5.32. In that case, *D* and *C* are spawned simultaneously, and should therefore be part of the same orthogonal component. However, this orthogonal component synchronizes on *C* and *E*, where *E* was spawned by a different orthogonal component. Using the basic mapping, the two forks cannot be independently mapped, as their interaction would need to be analysed, resulting in a different mapping to orthogonal regions. Therefore, we propose a more generic mapping, described next.

Our equivalent SCCD model consists of a set of orthogonal components, one for each activity diagrams construct. The order in which the orthogonal components are enabled, is defined by the condition that is present in the orthogonal component itself. Each orthogonal component will check whether it has the “execution token”, and if so, it passes on the

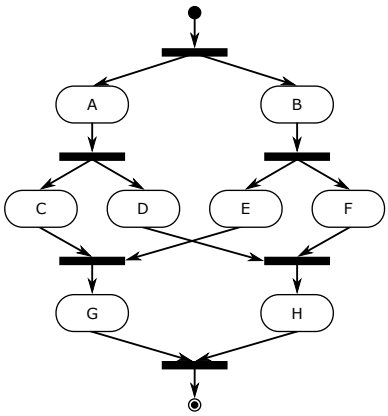


Figure 5.32: Problematic process model with a naive mapping to SCCD constructs.

token. All orthogonal components are executed concurrently, meaning that if suddenly multiple tokens exist, due to a fork, multiple orthogonal components can start their operation concurrently. Depending on the type of construct, the behaviour changes: activities execute and pass on the token upon completion, a fork splits the token, a join merges tokens, and a decision passes the token conditionally. We describe our transformation rules for each activity diagram construct in detail.

Transformation Rules

The following transformation rules are executed in the presented order. Before we actually start the translation, however, we first perform a minor optimization step: subsequent fork operations are merged into a single fork. This is not performed for performance considerations, but makes the mapping slightly easier. When a fork succeeds another fork, this is equivalent to the first fork also forking to the targets of the second fork, thereby bypassing the second fork. This optimization thereby removes two chained forks, allowing us to skip this case in the remainder of the mapping. While this pattern does not occur frequently, it must be taken care of, as it is a valid construct. The same optimization is performed for join nodes.

Combined with the previous section, in which we explicitly modelled activities using SCCD, it is possible to make the assumption that all activities have an SCCD representation. This is used to spawn new instances of the activity.

Optimization Figure 5.33 presents the optimization of fork nodes, as discussed previously. The first (topmost) rule makes sure that the first fork directly links to all targets of the second fork, removing the target from the second fork. This rule keeps the model semantically equivalent, as the second fork now has no successors. In the second (bottommost) rule, an empty fork node is removed, as it has no outgoing edges any more. This rule again maintains semantic equivalence, as the second fork has no successors left. Similar rules exist for the optimization of fork nodes.

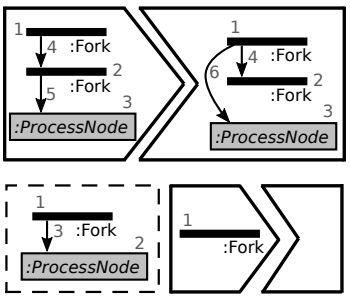


Figure 5.33: Optimize rules.

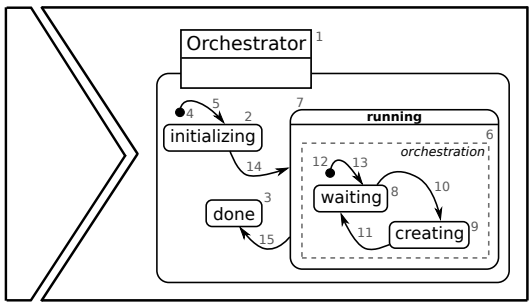


Figure 5.34: Orchestrator rule.

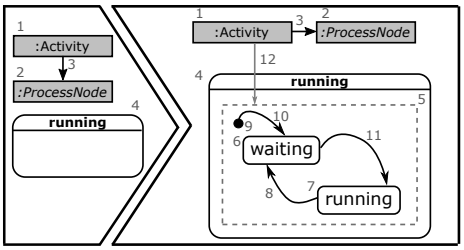


Figure 5.35: Activity rule.

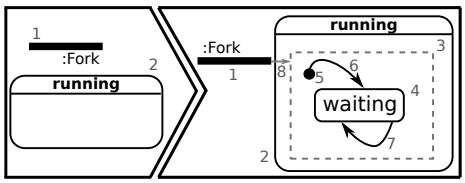


Figure 5.36: Fork rule.

Orchestrator Figure 5.34 presents the transformation rule for the orchestrator, which executes once. Each subsequent transformation rule extends a single composite state with an orthogonal region. The orthogonal regions execute all elements of the activity diagram in parallel, waiting for a condition to become true. The first step consists of creating the composite state and providing it with an orthogonal region that catches a spawn event, and performs the spawning of an activity. By defining this code here, it does not have to be reproduced throughout the other orthogonal regions, and maximising reuse.

Activity Figure 5.35 presents the transformation rule that executes for each activity. Activities are relatively easy to map, as they merely require the spawning of their associated activity (which, in our case, is modelled by another SCCD class). This is achieved by sending a spawn event to the orchestrator, and transitioning to a "running" state. We stay in this state until we have determined that the spawned activity has terminated, after which we mark the current activity as executed (*i.e.*, we pass on the token).

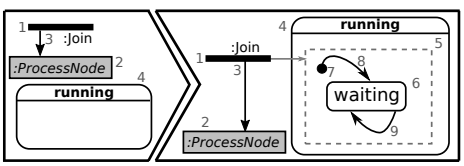


Figure 5.37: Join rule.

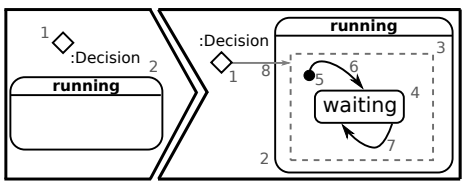


Figure 5.38: Decision rule.

Fork Figure 5.36 presents the transformation rule that executes for each fork node. Forking requires a single token to be distributed among all of its successors, without doing any computation itself. As such, our transformation rule adds an orthogonal component which continuously polls whether or not it has received the token. If it receives the token, it immediately passes the token to all of it successors simultaneously.

Join Figure 5.37 presents the transformation rule that executes for each join node. Joining is slightly more complex: it has to check for multiple tokens, before becoming enabled. When enabled, it consumes all of these tokens and passes on the token to its own successor, of which there is only one.

Decision Figure 5.38 presents the transformation rule that executes for each decision node. The final construct that we have to map, is the decision node. Similar to all previous nodes, we check whether we have a token to start execution. Depending on the input data that we receive, we decide to pass on the token to either the *true*- or the *false*-branch.

5.9.3 Evaluation

To evaluate our approach, we consider an FTG+PM for a simplified version of the power window, shown in Figure 5.39. This includes most aspects of the FTG+PM, such as concurrency (the 5-way fork in the beginning), decisions (is an error state reachable or not), and various types of activities (manual, transformations, action language, and external services).

Mapping this model using the previously presented transformation rules is simple, and results in the model shown in Figure 5.40. From the previous discussion, it became clear that this model does not look similar to the original FTG+PM model, as there might then be problems related to the concurrency. This SCCD model can now be executed directly, yielding identical execution results as if an operational semantics was implemented directly.

In future work, we plan to consider the benefits of combining service orchestration (activities modelled with SCCD) and the denotational semantics for process model enactment using SCCD. Indeed, as both the process and activities are modelled in SCCD, they can be combined into a single SCCD model. This single SCCD model can subsequently be analysed [217] or debugged [205], without any additional work. To achieve the valid and sound construction of this combined SCCD interaction/process model, composition rules of the single interaction SCCD model need to be investigated. Existing work on process-oriented inconsistency management in MPM settings [75] is a prime candidate to be augmented with such an approach. *Software Process Improvement* (SPI) techniques in general can greatly benefit from our approach as well.

5.9.4 Related Work

Most parts of related work are already touched upon in the previous section on service orchestration, as both topics are tightly interwoven. Most of these techniques rely on providing operational semantics to the used process modelling language. In our case,

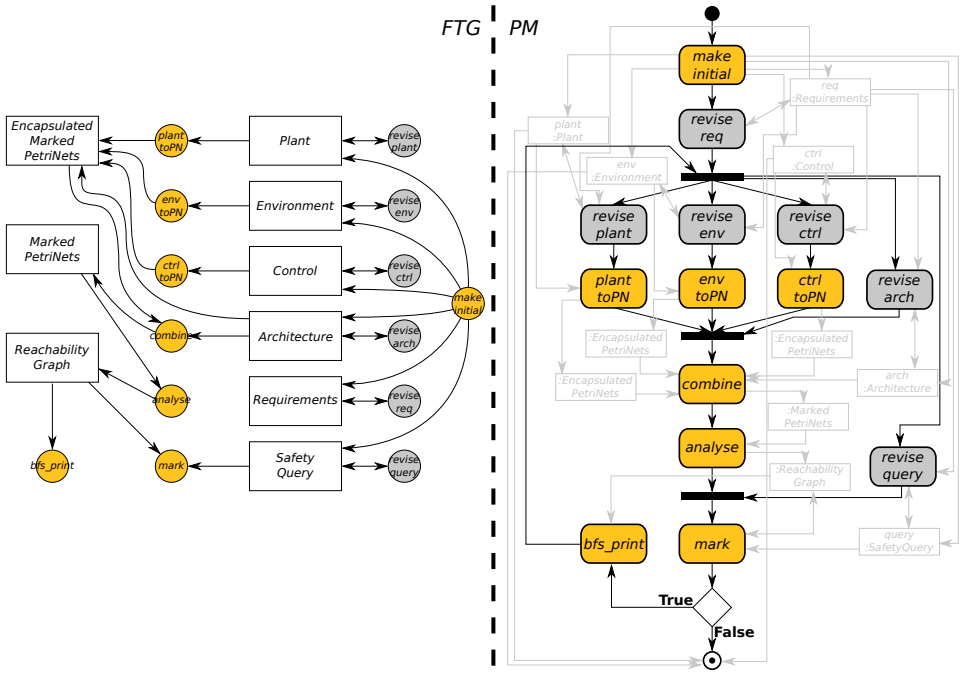


Figure 5.39: Power window FTG+PM.

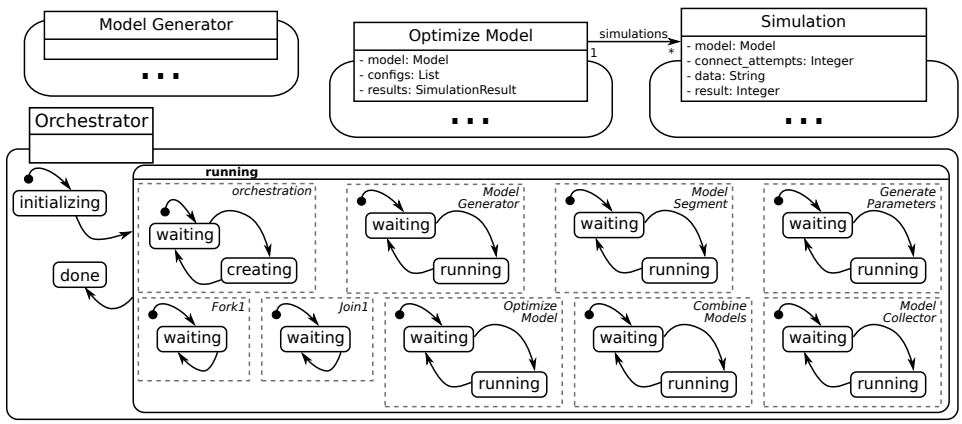


Figure 5.40: Power window FTG+PM from Figure 5.39 mapped to SCCD.

activity diagrams were used to define the process model. As this is a relatively popular language [229], several different forms of semantics are defined for this. In particular, the soundness and safeness of an activity diagram can be determined by mapping it to Petri Nets [287]. Both properties are then translated to the Petri Nets domain.

5.9.5 Link to Requirements

Support for the modelling and enactment of FTG+PM models influences **Requirement 3 (Process Modelling)**, **Requirement 4 (Multi-User)**, **Requirement 9 (Megamodelling)**, and **Requirement 10 (Portability)**.

Requirement 3 (Process Modelling) is influenced, as the presented approach offers enactment support for the FTG+PM. By modelling the semantics using model transformations, the semantics is also clearly defined, without relying on the implementation platform.

Requirement 4 (Multi-User) is influenced, as the PM is a structured way of supporting the interaction with multiple concurrent users, working on the same process.

Requirement 9 (Megamodelling) is influenced, as the FTG+PM model is effectively a megamodel: all data, activities, and formalisms are explicit models in the Modelverse as well. By offering enactment support, we show that megamodels can be fully supported.

Requirement 10 (Portability) is influenced, at the PM enactment can be seen as a platform-independent process execution language. As the enactment does not rely on details of the implementation platform, which are handled by SCCD, portability is achieved.

Summary

In the context of MPM, service orchestration and process model enactment is essential for the combination of multiple external tools. We propose an approach for handling the problems associated with implementing support for process model enactment: the importance of timing, reactivity, concurrency, and a potentially dynamic structure (due to multiple concurrent invocations) make it hard to code this behaviour. The process model is transformed into an equivalent SCCD model for execution, which has native constructs for timing, reactivity, concurrency, and dynamic structure. This preserves the modelling abstractions provided by activity diagrams, while gaining the execution of SCCD. In future work, the use of SCCD for both the activities and the process can allow for analyzability, as activities are no longer black box operations.

5.10 Action Language

Model transformations are often touted as the heart and soul of MDE [257], as it enables domain experts to define semantics themselves. And while often the case, some activities are not suited for a declarative formalism, making model transformations inappropriate. For example, constructing a reachability graph is intuitively simple with operational code, but is complex with model transformations, although possible. Many model management

operations are similar, and therefore profit from a procedural implementation. This procedural language forms the basis for the core library of model management operations, and can also be used to model activities.

5.10.1 Motivation

As procedural code is sometimes the most appropriate formalism, there is a motivated need. Indeed, MPM mandates to use the most appropriate formalism, which can be code. In this thesis, a new procedural language is created of which the semantics is explicitly modelled using graph transformations.

Why a new action language?

We have opted not to reuse existing general-purpose (programming) languages. While a plethora of such languages exist, some specifically tailored to a meta-modelling context, they rarely have a fully explicitly modelled syntax and semantics. Our new language offers several benefits that would otherwise have been difficult to achieve.

Existing general purpose languages, such as Python and Java, are huge: they have many syntactical constructs and their semantics is often not fully defined. For example, the semantics of Python is only defined through documentation and a reference implementation. Some standardized languages have a description of the semantics, but this description is not executable or complete (i.e., *undefined behaviour*). It is therefore difficult to explicitly model the language (e.g., for use in activities) and do meaningful operations (e.g., optimize code). The amount of features that these languages offer is also too generic for what we need it for: concepts such as generators are nice to have, but also complicate the design. Similarly, the extensive libraries that are offered in these languages should also be completely described in the language proper, or be formally defined. More restricted languages, such as OCL and EOL, are already better in this regard, but their syntax and semantics is even less formalized. Additionally, most aforementioned languages have only one interpreter, for example implemented in Java, restricting the implementation platform.

So while existing languages were considered, integrating them with the remainder of the application is troublesome without endangering our requirements. For these reasons, we have created a (minimal) new formalism for the definition of Action Code. This raises the question as to why we model the action language in the first place.

Why model the action language?

At the moment, most tools don't explicitly model their action language. Whenever a procedural language is required (e.g., as the action of a Statechart transition), this attribute is merely considered as a string. This string is then simply copied when generating code from the model, or is directly executed (e.g., using an *exec* statement) when the model is executed. Considering this situation, it is desirable to have an explicitly modelled action language for several reasons.

First, portability is a recurring requirement: it is difficult, if not impossible, to automatically translate existing languages (e.g., Python) to another language (e.g., Java). For example, when generating code from an SCCD model, the action on a transition should also be

mapped to the target language of the mapping. When languages like Python are used in the attribute, the target platform must also use Python or at least be able to integrate it. This means that code generation to platforms such as JavaScript become impossible. By modelling a minimal, neutral action language, which can be mapped to both Python and JavaScript, this problem could be circumvented. Second, if the action language and its semantics are explicitly modelled, debugging becomes easier [293]. All execution state is stored in models as well, and is therefore easily accessible for inspection and modification, while enforcing well-formedness. In the limit, activities can be defined to operate on action code and its execution state. Third, by modelling syntax explicitly, we can check whether a piece of code is correct before it is effectively executed. Furthermore, it can be enforced that some piece of code is to be a condition or is of a certain sub-expression type.

Why model the action language using graph transformations?

Modelling the action language requires three things: abstract syntax, concrete syntax, and semantics. We focus mostly on the semantics, as the (concrete and abstract) syntax is already done by most languages anyway (e.g., using a grammar), and we apply a similar approach. Given the graph-like structure of the action language abstract syntax, it is only logical to use graph transformations to define operations on this graph.

Using graph transformations offers several advantages. First, the graph transformation rules can serve as a visual piece of documentation, explaining the semantics of the action language. Second, graph transformations have a clearly defined (often formal) semantics that is relatively intuitive: nodes and edges are matched and then replaced. Third, our transformation rules can easily be used for code synthesis, searching for a pattern and rewriting it. Fourth, graph transformations are models themselves, and therefore action language semantics is again a model, with the usual advantages (mainly uniformity). Fifth, graph transformations are independent of the underlying implementation language, as there are no blocks of action code attached to the transformation rules.

For the visualization of the rules, we use a combined notation: a single subgraph is shown which has four colours. Black and blue elements are matched: these elements have to be present for the rule to become applicable. Red elements are negative application conditions: if these elements can be matched, the rule cannot be applied, even if all other applicable elements were found. Green elements are newly created after the matching phase was done. Blue elements are removed after they are matched. These colors therefore resemble the basics of CRUD operations: Create (green), Read (black and red), and Delete (blue). Updates are not supported, as none of the atomic elements can be updated.

5.10.2 Model

Modelling the Action Language requires to define its syntax and semantics. The syntax is used to create action code, while the semantics is used to execute it.

Syntax

As usual, the syntax of modelling languages is composed of both the abstract syntax (defining the structure and allowed constructs) and the concrete syntax (defining its visualization).

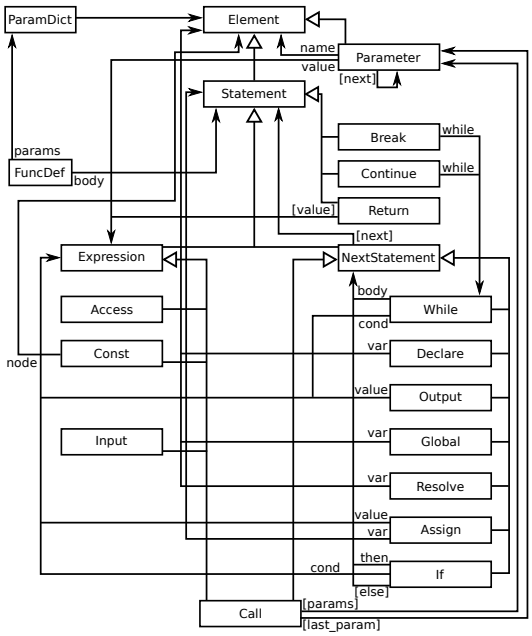


Figure 5.41: Abstract syntax of the Action Language. Associations whose name are within square brackets indicate optional associations. All associations have a maximum cardinality of 1.

Abstract Syntax The abstract syntax of the Action Language is shown in Figure 5.41, showing the atomic operations, such as *Declare* and *Assign*. For each of these constructs, additional constraints are defined, such as the maximum number of outgoing associations of a specific type. In our case, optional links are marked with square brackets (e.g., [value]), while all other associations are mandatory with exactly 1 instance. For example, it is shown that an *If* construct has exactly one association representing the condition, going towards an expression. As such, an *If* instance without condition expression is disallowed, and it is also disallowed for the condition to be a generic statement (e.g., a *While* construct). An *If* instance can, however, have an optional *else* block, but at most one. Due to its inheritance from *NextStatement*, there is also an optional *next* link to another operation. None of the elements have additional attributes defined on them.

This language is intentionally left as minimal as possible. For example, there is no *For* construct, as it can be emulated using a *While* loop. Nonetheless, it is of course possible to introduce such concepts, as long as a translation is defined which generates an equivalent model without the use of *For* constructs. Due to everything being modelled explicitly, this can be done using model transformations. Indeed, any possible language can be created, which only has to write out models using the Action Language abstract syntax, making it executable. This minimality makes it easier to provide support for the language (e.g., interpreter, debugger, JIT compiler) and forces more advanced concepts (e.g., list operations) to be modelled in the action code itself.

An example abstract syntax model implementing the Fibonacci algorithm is shown in Figure 5.42. This figure encodes the algorithm described in pseudocode in Algorithm 5.

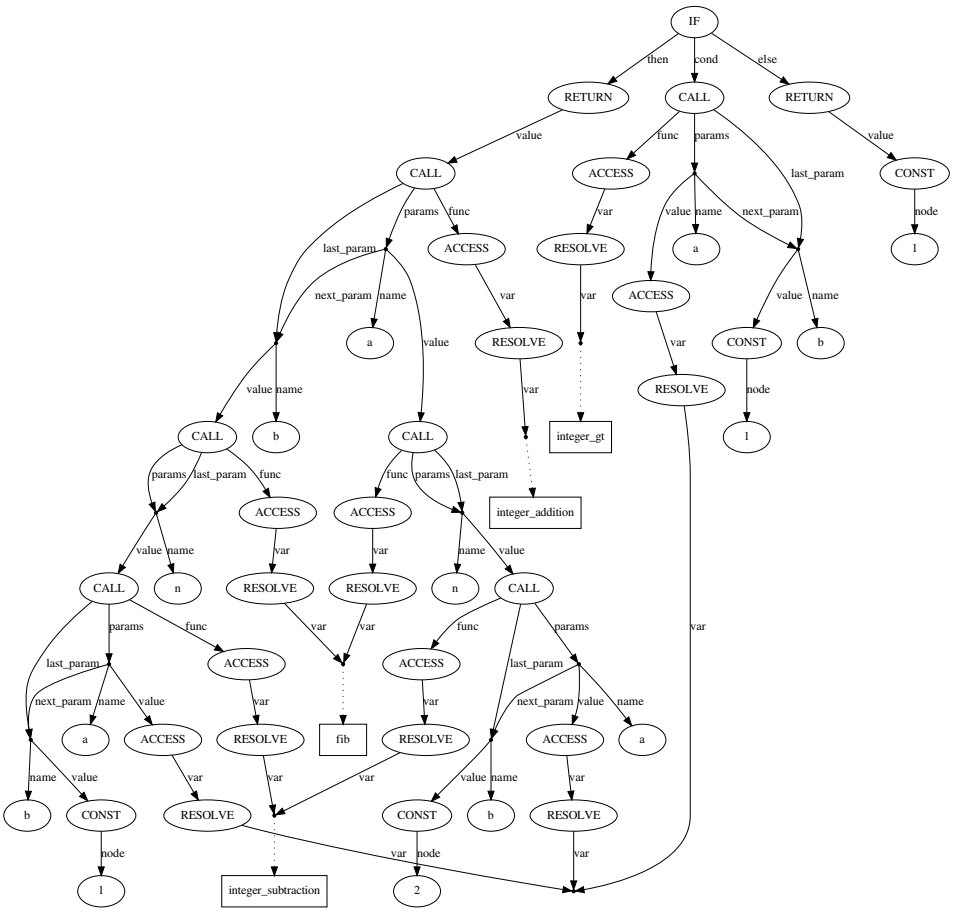


Figure 5.42: Abstract syntax of a Fibonacci algorithm written in the Action Language.

As all constraints imposed by the abstract syntax are valid, this model is a valid piece of Action Code.

ALGORITHM 5: Fibonacci algorithm implemented in Figure 5.42.

if $n > 1$ **then**
  return $fib(n-1) + fib(n-2)$
else
  return 1
end if

Concrete Syntax The concrete syntax of most programming languages is defined in the form of a grammar. This grammar is then used by a compiler compiler, which creates a parser for that specific language. The grammar used by our compiler compiler for the Action Language concrete syntax can be found in the source code.

For example, the concrete syntax of Algorithm 5 is shown in Listing 5.7. The meaning of the code is easy to understand, as the language bears similarities to existing procedural languages.

```
1 Integer function fib(n : Integer):
2   if (n > 1):
3     return 1!
4   else:
5     return (fib(n - 1) + fib(n - 2))!
```

Listing 5.7: Action Code for the Fibonacci algorithm shown in Algorithm 5

Interestingly, the Action Language has several additional primitives that are not usually found in a procedural language: each action language construct is itself a primitive in the Modelverse (prefixed with a "!"). For example, the construct *!If* can be used as a literal, indicating an instance of the *If* abstract syntax class. Action Language constructs are considered as primitives, as they are processed at the lowest level of the Modelverse, at a level where modelling is not yet defined. Using these constructs, it is possible to dynamically create or update action language without the use of modelling constructs.

The parser used is an external piece of software: we did not want to create a parser from scratch in the Modelverse, as that is out of scope for this thesis. Nonetheless, this parser is integrated in the Modelverse through the use of services (Section 5.8). This parser is unaware of the Modelverse internals and therefore does not send back action code directly, but commands. These commands are simple instructions to the Modelverse on which elements to create, and how to create them. Even if the internal details of the Action Language would change, the compiler can be left untouched. For example, an *If* construct is expanded by the compiler to a list of commands, in which the first one is the string "if", the second one represents the condition and is recursively processed, and so on. The Modelverse then processes this command to construct the necessary abstract syntax model.

Semantics

With the structure defined, the next step is execution. We rely on graph transformations operating over the primitive representation of this model: as a graph. This is required due to the Action Language semantics residing at the lowest level of the Modelverse, where there are no concepts of models and metamodels yet. It is therefore inefficient to rely on the modelling infrastructure: they reside at different levels in the Modelverse.

Semantics is instead defined operationaly. This operational semantics creates additional structures in the Modelverse, representing the execution frame. An execution frame includes an instruction pointer (to the model in the Modelverse), a symbol table (containing values for variables), a stack, and so on. By creating such an execution frame for each executing task, all information related to execution is explicitly present in the Modelverse. This explicit execution frame is used in the graph transformation rules, as some rules depend on the value of variables (e.g., *If* construct) and the state of the instruction pointer.

All graph transformation rules are shown and explained in Appendix B. Only a single rule is explained here in-depth, to give a basic understanding of the operation of the Modelverse. This rule is the execution of an *If* construct to go to the *else* branch and is shown in Figure 5.43. We consider three phases in the application of the graph transformation rule: matching, negative matching, and rewriting.

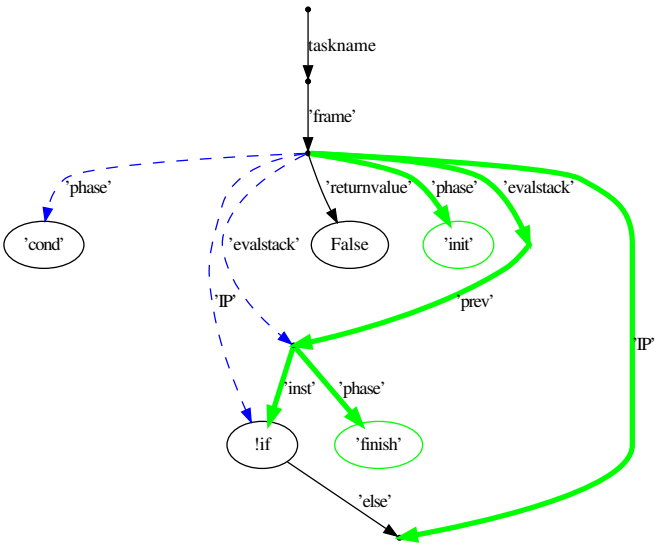


Figure 5.43: Graph transformation rule for the *If* construct to switch to the *else* block.

Matching First we look at the part that will be matched: the black and blue parts. These parts start from the Modelverse root (topmost node), from which we follow an edge dedicated to the specific task we are currently executing. The *taskname* is a parameter to the application of each rule and is set by the task manager (Section 5.11). For that task, the execution frame is read out, containing the instruction pointer (to *If*), the phase the instruction is currently in (*cond*, for condition), the return value of the last operation (*False*), and the current execution stack. The instruction pointer and phase are combined to determine when the rule is applicable. Only having an instruction pointer to the *If* is not sufficient, as we came back to this construct: the condition was already evaluated. This information is stored in the execution *phase*. A match is made if the return value is false, as the *else* branch is only executed if the condition evaluated to *False*. The evaluation stack is matched for later use, and is always present. Additionally, the matched *If* construct needs to have an outgoing *else* edge, as otherwise there is no *else* block. Not having an *else* block is fine, but then a different graph transformation rule is applicable: one setting the instruction pointer to right after the *if* block. If this rule is matched, we therefore know that the condition of an *If* statement (instruction pointer) has been evaluated (phase) to *False* (return value), and that the *If* construct has an *else* branch.

Negative Matching Now we determine whether or not the rule is still applicable given certain negative matches: the red parts. No negative match is given in this case, and therefore this phase is skipped. Another *If* rule exists which contains a negative matching part: the one executing this same instruction for when there is no *else* branch. In that case, the check must be made explicit, as otherwise multiple rules could be applicable simultaneously.

Rewriting Finally, we perform the rewriting if the matching was successful and the negative matching did not withold the application. Rewriting considers the blue (deleting) and green (creation) parts. First, all green elements are created. In our example, this creates a new instruction pointer to the target of the *else* link, and resets the phase to *init*, being the first phase of every instruction. Additionally, the execution stack is updated by pushing the *If* construct, with a phase of *finish*. This is required, as after the else branch has finished execution, execution falls back to the *If* construct, which then enters its final phase. After the final phase, control is passed on to the next instruction (not shown): either the one right after the *If* block or the enclosing operation (e.g., another *If* or function call). After creation, the blue elements are removed. These are the links for which an updated value has been stored in the stack (e.g., the phase). Therefore, the old phase and instruction pointer are removed, and the pointer to the execution stack is shifted to include the newly added element as well.

5.10.3 Evaluation

With the syntax and semantics of the Action Language defined, it remains to see whether it fulfills the goals we originally set out for in the motivation.

First, we turn back to the abstract syntax graph representation, as for example shown previously in Figure 5.42. Here, we see how easy it would be to perform optimizations on the abstract syntax graph through model transformations, as indeed the code is nothing more than a graph-based model. An example is constant folding for addition, shown in Figure 5.44. In this transformation rule, we search for a pattern that calls the *integer_addition* function on two constants, and replace this invocation with a constant. More involved optimizations are possible as well, for example if the source language is less verbose than the action language. For example, if the source language is identical to the action language, but includes a *For* construct, it is possible to automatically do this expansion using model transformations. That way we achieve the desired functionality of the source language, while maintaining the minimality of the Action Language and keeping the semantics of the language explicitly modelled. This kind of optimization is now often done in the compiler itself, for example through peephole optimization. The primary advantage of our approach is that the rule is easy to document and is intuitively clear, making it easier to maintain. Additionally, our approach works directly at the graph-level, making even complex optimizations, spanning huge portions of code, possible.

Second, it is possible to automatically generate documentation out of the rules, thereby providing a specification of the Action Language that is guaranteed to be up to date. In our case, we defined a model transformation which writes out the different matching rules to the format presented before (e.g., Figure 5.43). The output of this transformation is shown in the Modelverse technical report, where all graph transformation rules are shown and explained in detail. One such rule was shown in Figure 5.43 as an example.

Third, it is possible to automatically generate code from these transformation rules, thus providing an interpreter that is guaranteed to follow the specifications. We have defined this as a model-to-text transformation, which finds a *match set* of nodes, and based on that match, rewriting happens. Using this approach, it is possible to quickly port the interpreter to another platform, as most code is automatically generated. In the end, the automatically generated MvK consists of about 1250 lines of code, of which less than 100 had to be

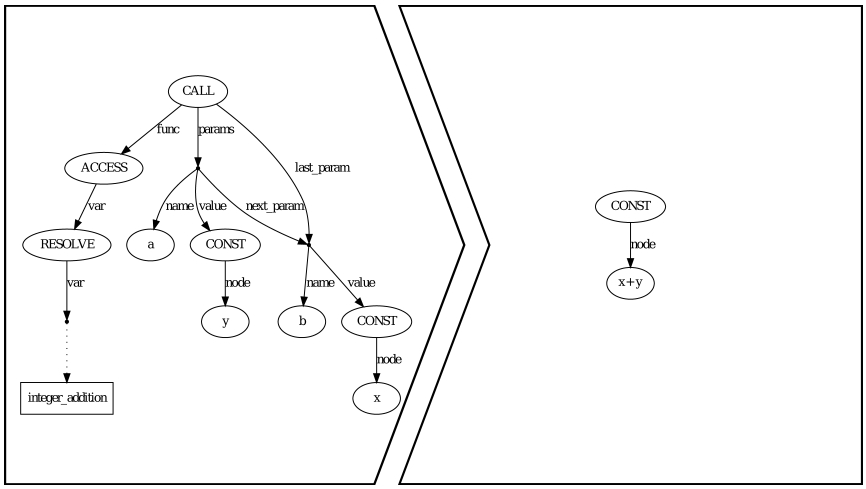


Figure 5.44: Example optimization rule for the Action Language: constant folding.

manually written. These manually written lines are the necessary preamble (imports, utility functions) and some minimal wrapping. Apart from the rules, primitive functions also have to be ported to the target language, though these are trivial to implement. For example, the `integer_addition` function maps to the + operator in Python. The Action Language interpreter can thus be ported to another language without much effort: the vast majority of code is automatically generated, and the remaining code is an intuitive mapping of primitive operations to the operations in the platform.

5.10.4 Link to Requirements

Explicitly modelling the syntax and semantics of the action language has an influence on **Requirement 2 (Activities)**, **Requirement 5 (Multi-Service)**, and **Requirement 10 (Portability)**.

Requirement 2 (Activities) is influenced, as activities can be implemented in whatever formalism is most appropriate for the job. In our case, the addition of an explicitly modelled action language allows activities to be defined in an operational action language as well. Apart from this, (declarative) model transformations are also possible.

Requirement 5 (Multi-Service) is influenced, as the Action Language compiler is implemented externally as a service. This immediately highlights that the Modelverse can make use of an external service, if the tool is too difficult to create from scratch in the Modelverse. Indeed, a parser is non-trivial to re-implement, making it more efficient to reuse an existing implementation.

Requirement 10 (Portability) is again influenced, as the semantics of the action language is modelled explicitly. Exactly the same action language semantics can therefore by synthesized for different platforms, from the same specification.

Summary

A new procedural language was created to serve as the Action Language in the Modelverse, with minimality and formal semantics as its primary goal. Both its syntax and semantics were explicitly modelled. For syntax, we presented the abstract syntax model and briefly mentioned how the concrete syntax was defined using a grammar. The explicitly modelled syntax allowed for more constraint type checking on action code fragments, which do not have to be persisted as text anymore. For semantics, we can automatically generate an interpreter for the language, including documentation, while offering access to all details of execution to debugging operations, such as the execution frame.

5.11 Task Management

As the Modelverse supports multiple users, each multiple concurrent connections to the Modelverse, some type of resource sharing becomes necessary for these tasks. We consider three types of resources: time, space, and access to memory.

For time, the operations of the various tasks have to be interleaved in the MvK, which does the computation. This is highly similar to process scheduling in operating systems. Other similarities can be found as well, such as managing files (models), users, processes (tasks), and memory management (MvS). For these reasons, we consider that the Modelverse implements the basics of a "modelling operating system".

For space, the models manipulated by the Modelverse have to be stored somewhere. We assume that the Modelverse will never run out of memory, as storage builds on top of a generic database system, for which advanced techniques exist that are out of scope of this thesis. Nonetheless, it is possible to impose a quota for the different users, thereby limiting the size of individual models, and limiting the number of models that the user can create. Such quota prevent a single user from using disproportionally more memory. Using the previously defined FTG, it is easy to store this data and enforce it, although this is done outside of the task manager.

For memory accesses, this is similar to time sharing, but at the MvS level, instead of the MvK level. Similar resource sharing is required at that level.

5.11.1 Motivation

Task management is therefore required in the Modelverse to implement resource sharing. For the Modelverse, we explicitly modelled task management using SCCD. As the primary advantages are related to development time and portability, this aspect is only briefly presented.

Why model task management?

As was the case for the GUI, the protocol wrapper, and the network protocols, the task manager interacts with many concurrent requests. Indeed, each of the connected tasks can send new input, receive new output, or perform computations at any point in time. Managing this complex stream of concurrent events is non-trivial with current programming languages. Ideally, resource management is identical among various implementations, meaning that all tools should implement the same policies.

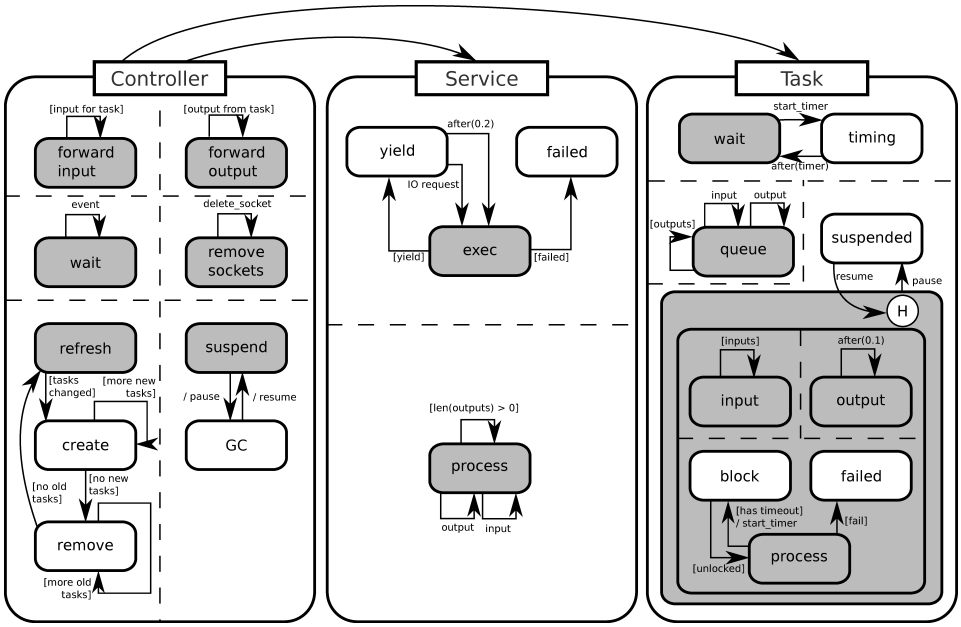


Figure 5.45: Task manager SCCD model (abstracted).

Why model task management with SCCD?

The high similarity with the GUI, protocol wrapper, and network protocol already suggests that SCCD might be appropriate to model the behaviour. And indeed, task management has the same sources of complexity, being the timed (e.g., timeouts of a client or a connection), reactive (e.g., new input or output), and concurrent (i.e., the various tasks running at the same time) behaviour. In this case, the dynamic structure of SCCD also comes in handy, as the various tasks all have their own statechart, which can be spawned or destroyed at runtime. Thanks to the code synthesis capabilities of SCCD, the model can again be used to generate code for several platforms.

5.11.2 Model

Figure 5.45 shows the SCCD model used to model the task manager. This is an abstracted view, which ignores actions and simplifies some conditions to keep the model readable. The full model can be accessed from within the Modelverse source code.

The SCCD model consists of three classes: Controller, Service, and Task. The Controller is the initial class, of which an instance is spawned in the beginning of execution. There are several concerns of the controller, but most importantly it should coordinate the different tasks and process input and output. Unused sockets are also pruned by the controller. Every so often, the controller queries the Modelverse for a list of active tasks. Tasks that were not yet known to the controller are spawned (spawns Task object), and tasks no longer present in the Modelverse are removed (remove Task object). Potentially, a task can also be a service, which is mostly the same, except that a Service has no attached computation.

From time to time, the controller suspends all tasks and issues garbage collection on the Modelverse State, thereby removing unconnected nodes.

The Task class combines computation with data processing and timing issues. First, the Task has a built-in timing structure: if the Action Language requests a timeout, this timeout is actually delegated to the SCCD model, where the pause happens. Second, the Task has a queue which processes all incoming and outgoing data until it can be processed by the computation. Third, the actual computation of the Task is left. Here, the task processes input events if there are some in the input buffer, and it checks for new output values every 0.1 seconds. Actual computation is done in the bottommost orthogonal component, where the computation can be blocked (due to a timeout), failed (due to an error), or processing. When the pause event is received, computation is paused and no operations happen, as the MvS is performing garbage collection. When the resume event is received, computation is resumed by using a history state.

The Service class is much more simple, as it merely processes input and output events. Services are similar to tasks, but have no attached computation: they are only used to store and forward data.

For each task and service running in the Modelverse, the respective class is instantiated. Thanks to the semantics of SCCD, each of these objects has an associated statechart, all of which run concurrently.

5.11.3 Evaluation

We now briefly evaluate the SCCD model created for the task manager. No detailed evaluation is given for this topic, as this does not present a significant contribution apart from development time and portability.

The task manager is highly similar to the other models we made using SCCD: all have to react to a stream of events. In the case of the task manager, we benefit from the use of SCCD by relying on its native constructs. For example, when a sleep operation is requested by the task, this is mapped to an *after* in the SCCD model of that task. This makes use of the most appropriate sleep operation for the current platform.

We make extensive use of the dynamic structure capabilities of SCCD, as we spawn several services and tasks dynamically. When a new task is detected in the Modelverse, the new task is managed by the MvK. We ensure that the Modelverse remains in charge (i.e., the Modelverse manages the active tasks). While executing a task, all tasks can concurrently process input and output, without bothering one another. This would otherwise only be possible through the use of threads.

5.11.4 Link to Requirements

Explicitly modelling task management has an influence on **Requirement 4 (Multi-User)**, **Requirement 5 (Multi-Service)**, and **Requirement 10 (Portability)**.

Requirement 4 (Multi-User) is influenced, as this task management model encodes how the interleaving between different users happens, to enforce (computational) resource sharing. The time sharing of different users becomes more intuitive to model through the use of timeouts and concurrent states.

Requirement 5 (Multi-Service) is influenced, as services also make use of the task manager, being similar to tasks. For each service, a service object is spawned in the MvK which forwards input and output events.

Requirement 10 (Portability) is influenced, as the model is again platform-independent. Specifically, the extensive use of concurrency and timing became much more portable by using these native constructs.

Summary

Task management is required to handle the resource sharing for the different tasks executing on the Modelverse. Resources include computation time, memory use, and memory access. We have primarily focussed on computation time, for which the resource sharing was explicitly modelled using SCCD. Thanks to this SCCD model, most of the accidental complexity of the timed, reactive, and concurrent behaviour was mitigated, while also automatically generating code for several platforms.

5.12 Performance

While performance is a non-functional aspect of an MPM tool, it remains an important consideration. Even more so, as the Modelverse serves many concurrent tasks simultaneously, using a client-server architecture over a (potentially high-latency) network. All in all, it is important to at least consider and evaluate the performance users can expect of our tool. Using a performance model, not only can we evaluate performance, but we can also perform what-if analysis and have efficient, repeatable benchmarks.

5.12.1 Motivation

A tool for MPM, such as the Modelverse, has to support a wide variety of features: interaction with multiple existing tools, multiple types of interface, multiple concurrent users, and so on. These high-level requirements result in several lower-level requirements. For example, the system must be distributed (to handle the "service" nature), must support concurrency (multiple users, each with their own domain of expertise), and performance must be monitored and optimized (to handle complex models). These low-level requirements are becoming more and more frequent in many types of software systems built today, although we will focus on these problems in the context of the Modelverse.

We will tackle the problems related to performance assessment and evaluation. As the Modelverse is concerned with performance, representative benchmarks are necessary. However, such benchmarks have limitations: they are executed on a shared-resource machine, meaning that performance results depend on the interleaving with other applications. For example, benchmarks executed while the anti-virus program is running or the system is being updated, result in incorrect performance results. Additionally, benchmark results strongly dependend on the hardware platform, such as CPU and main memory, but also the used network and all intermediate network components (e.g., switches). For example, benchmarks might be adversely affected if communication happens over a congested network, over a wireless network, or if the CPU is doing automatic frequency scaling. Benchmarks therefore have limited predictive power and repeatability: results are only valid for the machine on which they run, and in those exact circumstances.

These problems are caused by the non-determinism and non-configurability of the underlying platform. Performance evaluations are affected by the hardware used, as operations take different times to execute. The source of the problems therefore lies with timing. To address it, we do away with the current notion of time, shifting to a time over which we have full control: simulated time [124]. To use simulated time, we must make use of simulation, instead of executing the actual program. Therefore, we propose to explicitly model a Parallel DEVS [65, 338] model of the tool in question, effectively splitting the notions of wall clock time and simulated time.

5.12.2 Background: DEVS Modelling and Simulation

As was shown, DEVS modelling and simulation indeed has some advantages in this case. We now briefly elaborate on why we have used PythonPDEVS instead of other tools. PythonPDEVS is mostly interesting due to its balance between supported features [310, 313], compliance to the DEVS formalism [179, 313], and performance [308, 309, 313]. Additionally, the use of Python as the grafted language means that existing Modelverse functions could easily be integrated.

Other Tools

We present several other DEVS simulators, and mention how PythonPDEVS relates to them. Included tools were selected based on their popularity, (attributed) performance, or extensive set of features. We first give a high-level introduction of the tool and mention how it can be used, after which we discuss its features, compliance to DEVS, and performance.

Adevs Adevs [210] is a lightweight C++ library, offering DEVS simulation. Both atomic and coupled models are written in C++ code, which must include the Adevs headers. Due to the extensive use of templates, the headers contain all required source code. The simulation kernel and model are compiled into a single executable, and must therefore be recompiled after every model edit.

CD++ CD++ [328] is a DEVS simulator written in C++. Simulation of Cell DEVS models is its main feature, though normal DEVS models can be simulated too. DEVS models can also be coupled to Cell DEVS models. Atomic models are written in C++ and are linked into the simulation tool. Coupled models are written in a custom syntax, which is interpreted at simulation-time. Changes to atomic models require recompilation and linking to the simulation tool. Changes to coupled models don't require any recompilation at all, as these are interpreted during simulation. The complete behaviour of Cell DEVS models is defined using the custom syntax, which is completely interpreted. A graphical modelling environment, called CD++Builder [48], can be used to create the models. CD++ is a mature tool and is widely used in the literature for its Cell DEVS functionality [207, 259, 329].

DEVS-Suite DEVS-Suite [158] is the successor of DEVSJava [250]. Both are implemented in Java. Its features include visualization of coupled model simulation, event injection during simulation, and simulation tracking. Both atomic and coupled models

are written in Java and are loaded into the simulation tool through introspection. Changes require recompilation of the model, but don't require any action on the simulation tool. Both DEVS-Suite and DEVSJava are frequently used in the literature [62, 110, 216].

MS4Me MS4 Modeling Environment (MS4 Me) [258] is a DEVS modelling environment and simulator. It is written in Java and based on the Eclipse framework. Atomic models are created using a custom, natural language-like language called DNL, combined with fragments of Java code. Files are automatically translated to Java code, and subsequently compiled. Coupled models can be constructed using System Entity Structure (SES) [159, 336, 339], which are pruned before simulation commences.

PowerDEVS PowerDEVS [39, 162] is a Classic DEVS modelling and simulation environment implemented in C++. It consists of a graphical modelling environment, an atomic model editor, and a code generator. The code generator generates C++ code, which can optionally also be handwritten. PowerDEVS offers an intuitive modelling environment (with user-definable icons for models), combined with a library of models which can be reused, or used as examples.

PythonPDEVS PythonPDEVS [308] is a DEVS simulator written in Python. Due to its implementation in Python, an interpreted, dynamically typed language, fast prototyping of models becomes possible. Despite its interpretation-based nature, PythonPDEVS attempts to achieve high performance. Both atomic and coupled models are written in Python, making (re)compilation unnecessary. PythonPDEVS is used as the simulation kernel in several other tools. For example, DEVSimPy [58] offers a graphical modelling environment for coupled models, combined with an experimentation environment. A debugging front-end [296] offers a graphical modelling environment for atomic and coupled models alike, including advanced debugging capabilities.

VLE The Virtual Laboratory Environment (VLE) [226] is a multi-modelling and simulation platform written in C++. It includes an IDE for model development and experimentation. Models are combined in "projects", which are managed by an automatically created CMake script. Atomic models are written in C++ and thus require recompilation of the models after changes. The simulation kernel and IDE do not need to be recompiled. Coupled models are created using either the graphical environment (called GVLE), or by manually writing the XML files. VLE is the simulation kernel, with several bindings and "apps" to add functionality, such as an IDE (GVLE), distributed simulation using MPI (MVLE), Python bindings (PyVLE), and R bindings (RVLE).

X-S-Y X-S-Y [144] is a DEVS simulator written in Python. Its distinguishing feature is the verification of FD-DEVS (Finite and Deterministic DEVS) models. A small command line interface is provided, allowing for simulation control.

		ADEVS	*CD++*	*DEVS-Suite*	*MS4 Me*	*PowerDEVS*	*PythonPDEVS*	*VLE*	*X-S-Y*
Vendor	Pedigree	N	N	N	Y	N	N	N	N
	Documentation	Y	Y	N	Y	M	Y	Y	Y
	Support	N	N	N	Y	N	N	N	N
Model and input	Library	N	Y	N	Y	Y	N	N	N
	Coding	Y	M	Y	M	M	Y	M	Y
	Input	M	Y	Y	Y	M	M	N	M
Execution	Speed control	N	M	Y	Y	Y	Y	N	Y
	Multiple runs	Y	Y	N	N	Y	Y	Y	Y
	Batch runs	Y	Y	N	N	Y	Y	Y	N
	Parallel	Y	Y	N	N	N	Y	Y	N
	Distributed	N	Y	N	N	N	Y	Y	N
	Executable models	Y	N	N	N	Y	N	N	N
	Termination condition	Y	N	N	N	N	Y	N	N
Animation	Time Next	N	N	Y	Y	N	M	N	N
	State	N	Y	Y	Y	N	M	N	N
	Messages	N	N	Y	Y	N	M	N	N
	Transitioning	N	N	N	Y	N	M	N	N
	Sequence	N	N	N	Y	N	N	N	N
Testing and Efficiency	Tracing	Y	Y	Y	Y	Y	Y	Y	Y
	Step function	Y	N	Y	Y	Y	M	N	Y
	Verification	N	N	N	N	N	N	N	Y
	Backward clock	N	N	N	N	N	M	N	N
	Interaction	Y	Y	Y	Y	M	Y	N	Y
	Multitasking	Y	Y	Y	Y	Y	Y	Y	Y
	Breakpoints	N	N	N	N	N	M	N	N
Output	Delivery	Y	Y	Y	Y	Y	Y	Y	Y
	Graphics	N	Y	Y	Y	M	N	N	N
User	Orientation	N	M	N	M	M	N	N	N
	Financial	Y	Y	Y	N	Y	Y	Y	Y

Table 5.3: General evaluation, based on [208, 278].

Features

To evaluate the set of features, we distill a list of evaluation criteria from [208], adapted to the needs in DEVS modelling and simulation. Table 5.3 shows an overview of our evaluation results. A feature is either present (marked as a green "Y"), not present (marked as a red "N"), or only supported partially, with manual coding, or through the use of extensions (marked as yellow "M"). Normally, such a comparison is made using scores and weights [278], but this is omitted as this is highly dependent on the needs. More details can be found in [313].

Compliance

Another important aspect of a DEVS simulator, is how well it complies to the formalisms it supports. The original list of criteria for DEVS compliance [179] has been extended with additional Parallel DEVS criteria [313], and the need for DEVS model initialization [317]. Some tools support additional formalisms that are unrelated to this thesis and are therefore ignored. An overview is shown in Table 5.4.

		ADEVS	*CD++*	*DEVS-Suite*	*MS4 Me*	*PowerDEVS*	*PythonPDEVS*	*VLE*	*X-S-Y*
Formalisms	Parallel DEVS	Y	M	Y	Y	N	Y	Y	N
	Classic DEVS	N	Y	N	N	Y	Y	N	Y
Compliance	Translation functions	M	N	N	N	N	Y	N	N
	Event modularity	N	M	N	N	N	M	N	N
	Positive time	Y	N	N	M	N	Y	Y	N
	Select function	-	N	-	-	M	Y	-	N
	Confluent	Y	-	Y	N	-	Y	Y	-
	Initialization	N	M	M	M	M	Y	N	N

Table 5.4: DEVS-specific evaluation, based on [179, 313, 317].

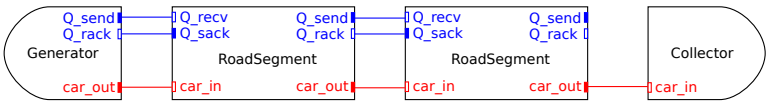


Figure 5.46: "Traffic" model, shown for 2 segments.

Performance

To evaluate the performance of all mentioned tools, a simplified version of a Kiltera benchmark [221] is used. The model does simple traffic simulation, where cars progress over a road, and slow down or speed up based on the cars in front of them.

The model itself consists of a generator, some road segments (processors), and a collector, as shown in Figure 5.46. After a randomly sampled time, a car is generated by the generator. The generator outputs the car and sends it to the connected road segment. Every road segment processes the car for a certain time (depending on the velocity), after which it is sent to the next road segment. A car can accelerate or decelerate, depending on their prefered speed, the speed limit of the road segment, and the cars in front of them. To prevent car collisions, road segments communicate with each other through the use of queries and acknowledgements. As soon as a road segment receives a new car, it sends a query to the next road segment, requesting whether the next road segment is free. It gets an acknowledgement back, stating how long it will take for the road segment to become available. The car at the current road segment will adjust its speed accordingly, depending on the maximal acceleration and deceleration values. If a road segment does not receive an acknowledgement in time, the car goes on to the next road segment without adjusting its speed. At the end of the road segments, a collector receives all cars and computes average velocity and average deviation from the prefered velocity. These statistics are used to test the correct implementation of the model in the various simulation tools.

Figure 5.47 shows the results of the Traffic benchmark. This shows the high performance of *ADEVS*, *VLE*, and *PowerDEVS*. PythonPDEVS comes in fourth, followed by the other tools. A more detailed analysis of the results can be found elsewhere [313].

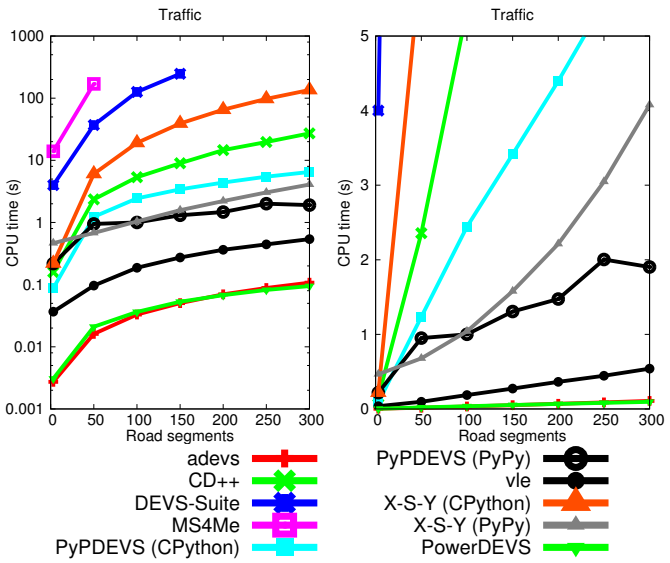


Figure 5.47: Benchmark results for the "Traffic" benchmark. The left figure uses a logarithmic scale, whereas the right figure is zoomed in on the fastest tools and uses a linear scale.

Summary

Given the provided results in terms of supported functionality, compliance, and performance, PythonPDEVS proved to be a well-balanced tool for our applications. Additionally, PythonPDEVS and the Modelverse rely on Python for non-model code, making it easy to reuse existing Modelverse code in the model.

5.12.3 Model

We now model the previously presented components using PythonPDEVS. We model the three main components of the Modelverse, as presented before: the Interface (MvI), Kernel (MvK), and State (MvS). For each component, we reuse as much of the original code as possible. As such, most execution logic (i.e., non-modal code) is simply imported and reused in the DEVS model, though some minimal DEVS wrapper code is necessary to make this fit in the DEVS formalism. By reusing existing logic, we are sure that the same behaviour is visible. The network inbetween all of these components must also be modelled: this is part of the environment that we have to model explicitly.

Modelverse Interface

The MvI was responsible for the creation of the high-level operations. While normally done by the user interactively, for example by clicking a button, we assume that this is provided as input to the simulation. The set of high-level operations to execute, is that of a power window modelling and safety verification [316].

Model Due to the previously made restrictions, the DEVS model of the MvI is rather simple. It contains a list of operations to execute, and a set of operations to execute for specific models (e.g., details on how to model a control model). Whenever an operation is sent, we wait for a reply before sending the next request. While usually the reply is presented to the user, it is now thrown away, as we operate in batch mode. Despite this simple explanation, the DEVS model is quite convoluted, as we have to implement an autonomous MvI from scratch. For each response that the MvI receives, it checks whether it has to stop the simulation. This is done by setting a state variable, which is used by the termination condition.

Calibration As there are not many parameters to the MvI, not a lot of calibration was required. The list of operations to be executed, however, needs to be defined. As previously mentioned, we used the power window case study, for which we needed the list of high-level operations. To obtain the list of actual requests used, a real run of the Modelverse was instrumented with logging, thereby logging the various high-level operations sent to the Modelverse. This resulted in a set of no less than 16,000 requests, grouped in 75 "request blocks", as many requests don't depend on the result of the previous ones. A single group is sent simultaneously, thereby reducing the round trip time, as it is also done in the real execution of the Modelverse.

Modelverse Kernel

The Modelverse Kernel was responsible for the computations: converting the high-level model-management operations to low-level graph operations. As we reuse much of the logic from the actual Modelverse code base, this model is mostly a wrapper around that logic. Nonetheless, task management had to be reimplemented to suit the DEVS formalism.

Model The DEVS formalism requires to have full control over timing behaviour. This did not suit well with the original specification of the task management of the MvK, which was modelled with SCCD [298], instead of coded. While this is ideal for execution, generated SCCD code contains timing information and threading, making it unusable in a DEVS model. All task management was reimplemented in DEVS, with behaviour similar to the original implementation. Most of the other code, specifically the translation code, was reused.

Calibration The MvK needs some calibration, as the computations take time. This was not needed in the MvI, as we abstracted away the time needed by the MvI to come up with the next high-level operation. We cannot do this for the MvK, as there are be many steps involved in translating the high-level operations to low-level operations. We have implemented two types of interpreter: one based on pure interpretation, and one based on a Just-In-Time (JIT) compiler. The pure interpreter will take less time to do the conversion to low-level operations, but will generate more low-level operations. On the other hand, the JIT will take much more time for the conversion, but generates significantly less operations. Results were measured by doing a first DEVS simulation run, where the computation was actually performed, and the time measured. This gives a large set of samples: $34,638,621$

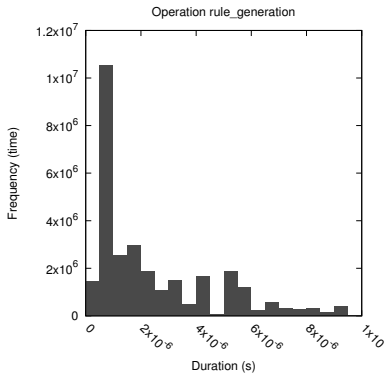


Figure 5.48: Distribution of time taken for rule generation in the MvK.

operation	samples	averages
read root ID	1	0.00000286s
read dictionary	13,456,485	0.00000111s
read dictionary keys	181,314	0.00000804s
read node value	14,538,084	0.00000034s
read dictionary by node	491,026	0.00000179s
read dictionary edge	503,147	0.00000289s
create node	775,370	0.00000068s
create value	2,157,118	0.00000092s
create dictionary	1,329,637	0.00000452s
delete edge	1,489,940	0.00000288s
delete node	31,092	0.00000854s
dictionary key lookup	9,093	0.00001803s
create edge	5,263,314	0.00000184s
read outgoing edges	4,627,855	0.00000150s
read incoming edges	1,325,308	0.00000178s
read edge	3,913,127	0.00000039s
garbage collect	21	0.88014233s

Table 5.5: MvS operations and the measured calibration results.

to be precise. For the JIT, the distribution of these values is shown in Figure 5.48. Most operations take a relatively short time (compiled operations), and some take significantly more time (interpreted operations and compilation). For the pure interpreter, all invocations are nearly instantaneous.

Modelverse State

The Modelverse State was responsible for state manipulations. Again, we reuse almost all logic from the graph database that underlies our implementation, and therefore the DEVS model for this is merely a wrapper which monitors the time the operation takes.

Model The DEVS specification of the MvS is trivial: when receiving a list of low-level operations, the operations are executed in order, and the simulation time is incremented with the time it takes to execute the operations, which is sampled from a distribution. Results are subsequently sent back to the MvK.

Calibration Calibration is again required to determine how long each of the low-level operations takes. For this, the same approach is used as with the MvK: first do a "calibration run", in which timing information for each operation is recorded. Table 5.5 presents how many samples were made for each of the low-level graph operations, and gives some information on the values found. Some operations are not executed that frequently, as the JIT was used, thereby lowering the number of low-level operations.

Network

Apart from the Modelverse components, our DEVS model also requires an explicit model of the network. While normally the network is used as-is, for example through the use of sockets, we now explicitly model this, giving us full control. This DEVS model only

has to model latency and bandwidth. Other network characteristics, such as routing and packet loss, can be ignored, as the original application also relied on TCP/IP to handle these aspects. As we are not interested in any of these characteristics in particular, we can abstract these away in this model.

Model The DEVS model of our network is relatively simple, and only models latency and bandwidth. For each incoming message, we delay the sending by a time that is equal to the latency (a fixed cost), and add the time due to the restricted bandwidth. As such, the message is serialized using JSON, which results in a string that has to be transferred over the network. This string has a number of characters, and thus a number of bytes, as we use ASCII encoding. Using this information, and the maximum bandwidth, we can compute the time it takes for the message to pass over the network.

Calibration For the network, calibration is again required, as we need to find values for the latency and the allowed bandwidth. Tools like *ping* can be used to estimate the round trip latency, and tools like *iperf* can estimate the network bandwidth. When two components are ran on the same machine, latency and bandwidth can be abstracted to zero and positive infinity, respectively. After the various operations were calibrated, the model was validated against the running Modelverse that was used for calibration.

5.12.4 Evaluation

We now apply this model for our original purpose: performance evaluation. We consider four aspects of performance analysis that are influenced: what-if analysis, benchmark automation, determinism, and benchmark performance.

What-if Analysis

First is the possibility for what-if analysis. While with usual benchmarks, we need to have access to the benchmarking hardware to generate meaningful results, a what-if analysis allows us to use a hypothetical set-up. This makes it possible to prototype specific hardware, such as a low-latency network, without actually having to invest in it beforehand. Indeed, it might not be worthwhile the investment if the benchmark results prove insatisfactory. Additionally, we can explore a set of possibilities, optimizing the total cost.

With a DEVS model, what-if analysis becomes possible. For example, we can easily change the network latency parameter from $0ms$ (e.g., local execution) to $500ms$ (e.g., satellite connection), and see the effect. Figure 5.49 and Figure 5.50 present results for varying network latencies between the MvI and MvK, and the MvK and MvS on the total execution time, respectively. These results indicate that we can increase the latency between the MvI and MvK without too many problems, which is one of the requirements of the Modelverse: it must be able to run distributed, with MvIs running on various machines, possibly even over the internet (i.e., relatively high latencies). Although the execution time does increase, it less than doubles when going from an instantaneous connection ($0ms$ latency) to a high latency connection ($> 100ms$ latency). The MvK and MvS, however, should ideally be ran with very low latencies, as the results indicated: execution time is highly dependent on this

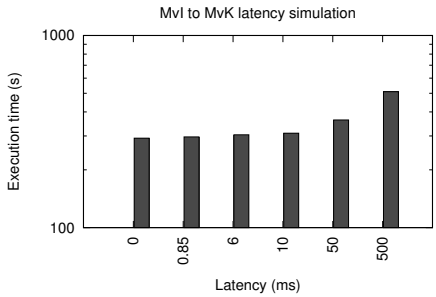


Figure 5.49: Influence of MvI - MvK latency.

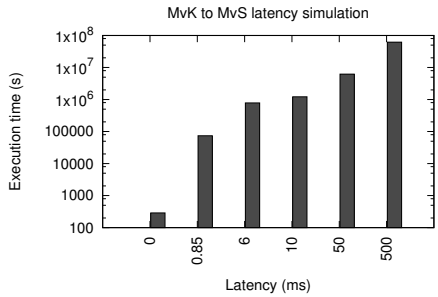


Figure 5.50: Influence of MvK - MvS latency.

latency. While not ideal, this does not form a significant problem: the MvK and MvS are two server-side components, and are likely hosted close to one another.

These results are as expected, though they still provide added value: we can quantify how long it will take, up to some degree of certainty. For example, we can now state that the performance is tolerable, even when using a high-latency connection between MvI and MvK. For performance critical software, hard restrictions are often imposed on the acceptable delays, rendering quantification important. Similarly, for the MvK and MvS we intuitively know that splitting them is not a good idea performance-wise, as they communicate a lot. Nonetheless, through simulation we can quantify the expected execution time, allowing for a trade-off. Constraints and costs can be combined with the execution time to find a cost-optimal solution.

Notwithstanding these advantages of the DEVS model, it remains an abstracted simulation. As such, results will not be perfectly accurate, though rather to be within some bounds around the actual values. The more fine grained the model becomes, the less abstractions are made, and the more accurate the results become. However, less abstractions will also increase the simulation time. For our purposes, the current level of abstraction proved sufficient.

Automation

A second advantage of using simulation is that everything can be fully automated. Going back to our previous example of network latency, it is immediately obvious that we cannot automatically switch the system over to various types of network, even if we were to have all the required hardware. More complex hardware changes, such as disconnecting a network cable, or swapping the CPU, are even more difficult to do automatically. As such, these operations are done manually, introducing manual intervention and thus non-determinism: the point in time when the cable is manually disconnected, will vary between two simulation runs, even if only by a few milliseconds.

With a DEVS model, all these operations become trivial to execute, as all relevant aspects of the system, such as the network, are modelled explicitly. Disconnecting a network cable is then just sending an event to the network model, which subsequently no longer transmits messages. All results presented in this paper were obtained completely automatically, and without the use of any platform-specific operations. Additionally, by disconnecting from

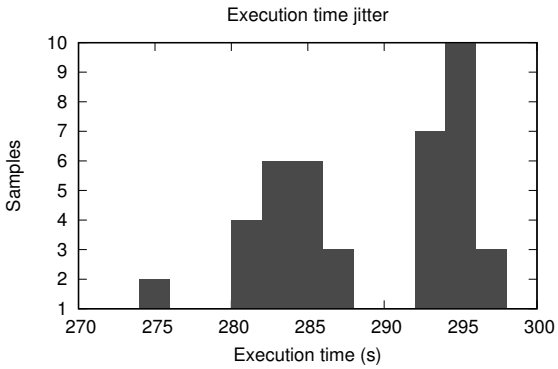


Figure 5.51: Actual execution results.

the platform, it becomes possible to run multiple simulations simultaneously, as all platform changes are simulated as well.

Determinism

A third advantage of using simulation is determinism. Normal benchmark execution, on actual hardware, has many causes of non-determinism: a shared-resource machine, a shared network, varying network characteristics, and so on. While it is possible to replicate benchmark results, they are only identical to some level, and it might take several tries. If execution takes a significant amount of time, we often do not want to execute the same program multiple times. Additionally, it makes it difficult for others to replicate results.

With a DEVS model, determinism is achieved by modelling all sources of non-determinism, such as the network and computation times, explicitly. Changes to the setup can be evaluated with a single simulation run, which then automatically reuses *exactly* the same simulation setup, as all data is deterministic. Not only will this result in exactly the same behaviour, but it will also immediately show the impact of changes on performance, without non-deterministic jitter on the results. Of course, the performance effect *in general* still requires multiple simulation runs with varying seeds. Figure 5.51 presents the result of several actual execution runs of the program. While there is less than 10% difference between all executions, the difference is noticeable. With simulation, results are always equal, given the same random seed.

Performance

A fourth advantage of using simulation is that simulation is often faster than executing the actual application. During execution of the program, only a part of it is spent on actual computation, as much overhead exists: task switching, network timeouts, network transfer delays, and so on. During simulation, mostly the same code is still executed as in the original application, though we can skip many of these sources of overhead. For example, network latency only affects simulated time, as the network is not actually used, but only simulated. As such, high network latencies (e.g., $500ms$) will not take much

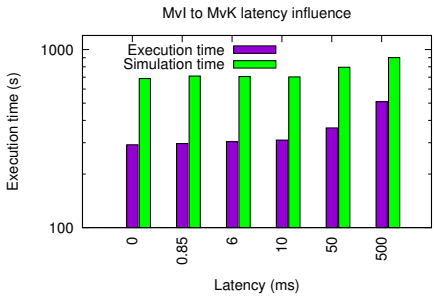


Figure 5.52: Varying MvI - MvK latency.

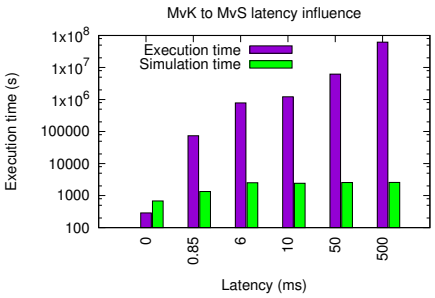


Figure 5.53: Varying MvK - MvS latency.

longer to simulate than no network latency at all (i.e., $0ms$). Similarly, simulations can happen on fast computers, while using the time that a slow computer would process on it. This is advantageous, as the wall clock time then progresses slower than the simulated time, meaning that simulation becomes faster than actual execution. Notwithstanding these speedups, simulation also introduces its own types of overhead. While this is not to be ignored, the overhead induced by the network is generally larger than the simulator overhead.

Figure 5.52 presents the difference between the time needed for simulation, and the simulated execution time for a range of different latencies between the MvI and MvK. In this case, simulation is always slower than actual execution, as this network connection is not frequently used. This is comparable to the results we had before, where we saw that the influence of this latency on execution time is minimal. We do, however, notice that the simulation time also increases, which was against expectations. It seemed that the JIT compiler, which is a core component of the MvK, reacts differently depending on how much time it takes to execute some functions.

Figure 5.53, on the other hand, shows the results for varying the MvK to MvS latency. Here, the simulation time also increases, though far less than when executing the actual system. In this case, simulation clearly outperforms actual execution: for $500ms$ latency, simulation is more than $24,000$ times faster.

5.12.5 Related Work

The traditional approach to performance evaluation is the creation of benchmarks, which execute the application and measure how long it takes. While this yields trustworthy results, there were many potential problems, as mentioned in this paper. DEVS modelling has been used before to measure the performance of software applications, for which we now give some examples. One DEVS model was for a distributed DEVS simulator [272], where the DEVS abstract simulator was modelled using DEVS itself. This was primarily done to monitor the behaviour of the simulator in exceptional cases, such as when a disconnect happens. Not many details were given on the calibration of the model, and the DEVS simulator itself was not performance-critical in this paper: it was a minimal distributed DEVS simulator without advanced synchronization protocol. As such, no attention was payed to the performance of the application. Another DEVS model is that for a new algorithm for property-based locking in collaborative modelling [87]. Here,

DEVS was also used for the context of (distributed) execution, though the focus was not on execution performance, but on deterministic simulation as to whether a lock was granted or rejected.

5.12.6 Link to Requirements

Explicitly modelling the performance has an influence on **Requirement 2 (Activities)**, **Requirement 4 (Multi-User)**, and **Requirement 10 (Portability)**.

Requirement 2 (Activities) is influenced, as the DEVS model can be used to optimize the performance of the execution of activities. It does not directly influence the possibility of having activities, though their efficiency is crucial to their usability.

Requirement 4 (Multi-User) is influenced, as the performance of resource sharing can be analyzed using a DEVS performance model. Various situations can be tested, and possibly different algorithms for resource sharing can be compared in a meaningful way.

Requirement 10 (Portability) is influenced, as performance normally significantly depends on the platform. If a different platform is used, performance results can often not be reproduced. Even when the same platform is used, different benchmark runs will have some jitter due to the other load present on the platform or due to external non-deterministic behaviour, such as network latencies.

Summary

We created a performance model of the Modelverse in the Parallel DEVS formalism, thereby splitting wall clock time and simulated time. This makes performance analysis and optimization easier and less ad-hoc: we achieve full control over time. This allows for what-if analysis, deterministic and fully automated benchmarks, and potentially faster benchmark results. Results can also be applied for performance optimization. For each aspect, we described how our DEVS model addresses these problems, resulting in the aforementioned advantages. We believe that this approach is similarly applicable to many other complex, distributed applications.

Summary

We have described all aspects that make up the Modelverse, and shown how they are modelled. While all aspects were modelled (to some degree), several of them provided not too many benefits apart from faster development time, portability, and increased understandability of the application (e.g., documentation). These are the usual advantages attributed to modelling. Other aspects proved more worthwhile to model explicitly, as their explicit models enabled additional benefits in several domains related to MPM. These form the primary contributions of this chapter. Explicitly modelling the conformance algorithm proved useful to enable meaningful model exchange between different tools and allowed for multiple conforming metamodels simultaneously [314]. Explicitly modelling the physical type model in the linguistic dimension proved useful to make models and algorithms independent of implementation details, allowing it to be changed at runtime. Explicitly modelling the interaction with external service using SCCD proved useful to

integrate (multiple) existing external black-box tools while retaining full control and analyzability over its behaviour [297]. Explicitly modelling the enactment of an FTG+PM by mapping it to SCCD proved useful to rely on existing operational semantics, while potentially allowing for increased analyzability when combined with explicitly modelled activities [297]. Explicitly modelling the action language proved useful to integrate code fragments in existing models and meaningfully operate on existing code fragments, while also significantly increasing portability of the Modelverse and all generated models [304]. Explicitly modelling the performance proved useful to allow for what-if analysis and reduce execution jitter, but also to decrease the time needed for benchmarks. Even though each component and all mentioned benefits could equally well have been achieved without the use of explicit models, it would likely be harder to achieve.

Chapter 6

Modelverse as a Foundation for MPM

To illustrate the value of the Modelverse for further research in MPM and to show that it is applicable to the concerns of the three types of users considered, several contributions are built on top of our tool. Each contribution can be considered stand-alone in the domain of MPM, and is often published as such. All contributions rely on MPM, thereby relying on the Modelverse, and are often valuable in the context of MPM as well, thereby being a contribution to MPM. We provide a motivation as to why this contribution is useful, explain the approach, and subsequently evaluate it. We consider three contributions, one for each type of user considered: the modeller (Section 6.2), the language engineer (Section 6.3), and the Modelverse tool developer (Section 6.4). First, however, we present how the Modelverse addresses the Power Window case study, thereby illustrating its full support for MPM (Section 6.1).

6.1 Power Window Case Study

To indicate that the Modelverse is capable of full support for MPM, we first show how it handles the enactment of the FTG+PM for our power window case study. This FTG+PM, previously introduced in Figure 2.22, but repeated here in Figure 6.1, is modelled explicitly in the Modelverse. At the left hand side, the FTG presents the different formalisms used (*e.g.*, Plant, Environment, ReachabilityGraph), as well as all operations between them (*e.g.*, Plant2EPN, Combine, Analyze, Mark). At the right hand side, the PM presents the order in which these operations are defined, as well as the specific modelling artefacts that are propagated between operations (*e.g.*, Combine combines the Encapsulated Petri Nets which originate from the Plant, Control, and Environment, together with the architecture model). The flow of the problem at hand is easy to deduce: domain experts create models for the plant, control, environment, architecture, and safety query in a domain-specific language, following the requirements defined beforehand. The plant, control, and environment model are individually translated to Encapsulated Petri Nets (*i.e.*, Petri Nets with ports), which are merged together with the architecture model. The resulting Petri Net is analyzed for reachability, on which the safety query is executed. If the query is found, an unsafe situation

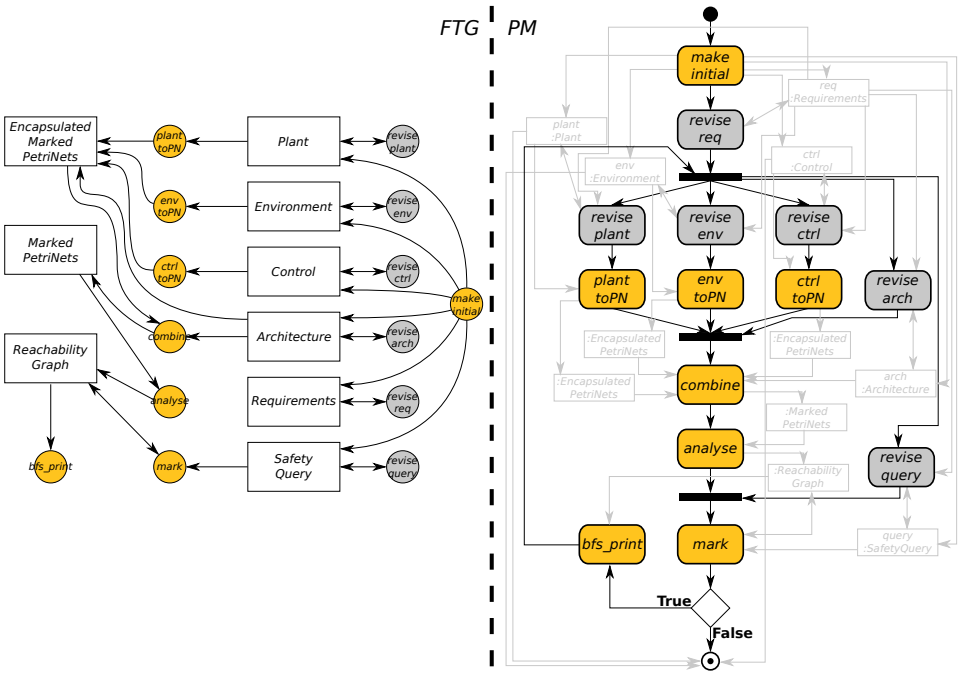


Figure 6.1: FTG+PM of our example: the development and verification of a simplified power window.

can be reached and a counter-example is presented in the form of a series of operations. Users are then prompted to make revisions to their models. If the query is not found, the system is deemed safe, and the process finishes.

We consider the various aspects of MPM, as reflected in our requirements, and how they are supported by the Modelverse. While we could not guarantee that this list of requirements was complete, the following evaluation on a case study shows that these requirements are at least sufficient.

6.1.1 Requirement 1: Domain-Specific Modelling

Multiple domain-specific languages were implemented in the use of the Power Window case study. An overview of the various domain-specific models and languages is shown in Figure 6.2. At the topmost level, we see ClassDiagrams as the meta-circular level. Below, we see all formalisms used in the example, which were all created through language engineering. At the lowest level, all instances of the engineered languages are shown.

For example, the metamodel of the plant language is shown in Figure 6.3, containing the different concepts that can be used: two kinds of state, and three kinds of transitions. Similar metamodels are present for the other languages as well.

These different domain-specific languages were subsequently used to model each part of the system: the environment (Figure 6.4), safety query (Figure 6.5), control (Figure 6.6),

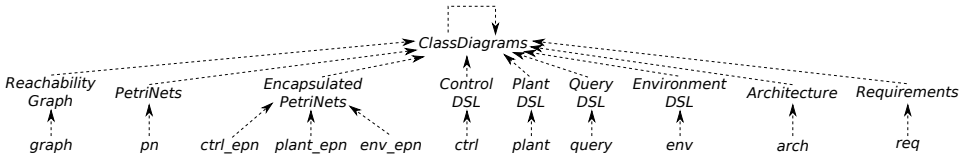


Figure 6.2: Meta-modelling hierarchy of the example.

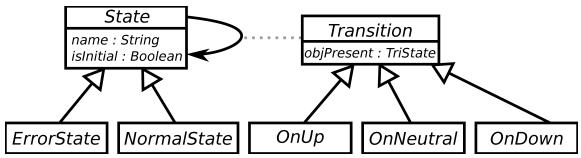


Figure 6.3: Plant metamodel, describing allowed constructs for a plant model.

plant (Figure 6.7), and architecture (Figure 6.8). These models are compact and provide a much more conceptually clear overview of the parts of the system, than if it were to be done in a GPL. Additionally, these models are closer to the problem they are modelling and require no knowledge of programming.

6.1.2 Requirement 2: Activities

For all domain-specific modelling languages made before, activities were defined to map them to the semantic domain of Petri Nets. Using all these Petri Net models, which are woven together, a single Petri Net is constructed that can be verified using the safety query.

Many approaches exist to model activities, though the Modelverse implemented model transformation using RAMification [171] and a procedural action language. In the spirit of modelling all activities using the most appropriate formalism, we present some different activities used in the Power Window case study.

Translating domain-specific languages to more general purpose languages is often done through model transformations, as we can remain at the domain-specific level. An example model transformation rule is shown in Figure 6.9, where the environment model is mapped to a Petri Net. Note the presence of traceability links (label 7 and 8), which will be mentioned again later on. Model transformations are the most appropriate for this translation, as model transformations automatically find all possible combinations. For example, the two matched events (A and B) might not be the only ones in the group, and there are actually two ways to switch between them (from A to B, and vice versa). Using

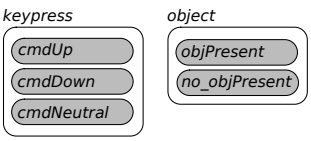


Figure 6.4: Environment model.

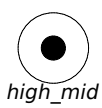


Figure 6.5: Safety query model.

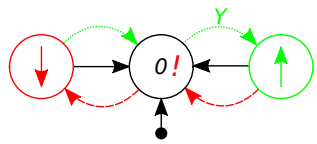


Figure 6.6: Control model.

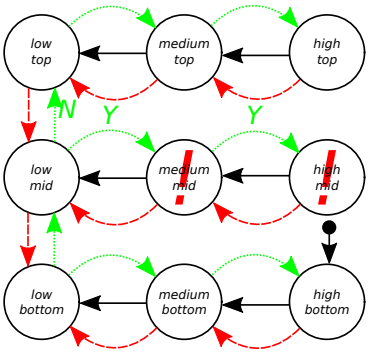


Figure 6.7: Plant model.

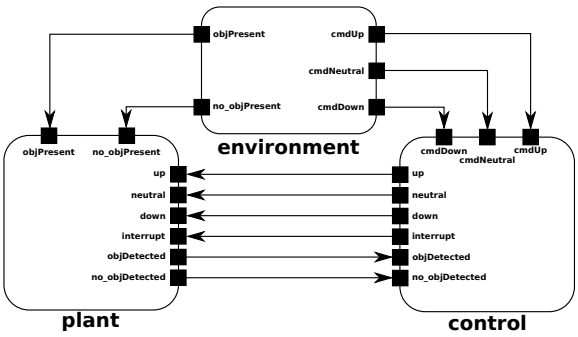


Figure 6.8: Architecture model.

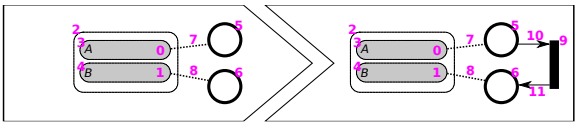


Figure 6.9: Rule for transforming the environment model to an encapsulated Petri Net.

model transformations, all cases are immediately taken care of using only a single, intuitive transformation rule.

Merging together the different models is ideally done using action language in this case, as it is mostly a retyping operation. Indeed, we only have to merge different models into a single model, which is automatically done by the Modelverse, but then have to retype everything to the output metamodel. While this could be done with model transformations as well, we would have to define several rules (mapping nodes, mapping links) for each input metamodel. Using a procedural action language, this only takes a few lines of model management code, as shown in Listing 6.1.

```
include "primitives.alh"
include "modelling.alh"

Boolean function main(model : Element):
    Element keys
    String key
    Element split

    keys = dict_keys(model["model"])
    while (set_len(keys) > 0):
        key = set_pop(keys)
        split = string_split(read_type(model, key), "/")
        retype(model, key, string_join("Encapsulated_PetriNet/", split[1]))

    return True!
```

Listing 6.1: Merging activity for Encapsulated Petri Nets.

In the FTG+PM, no difference is seen between these two types of activities, as they are completely transparent to the user executing them. A third type exists, being manual activities. These are highlighted in black in the FTG+PM, and have to be manually executed by the modeller. In the Modelverse, when executing a manual activity, the modeller is presented with a limited modelling environment where the specified model can be modified.

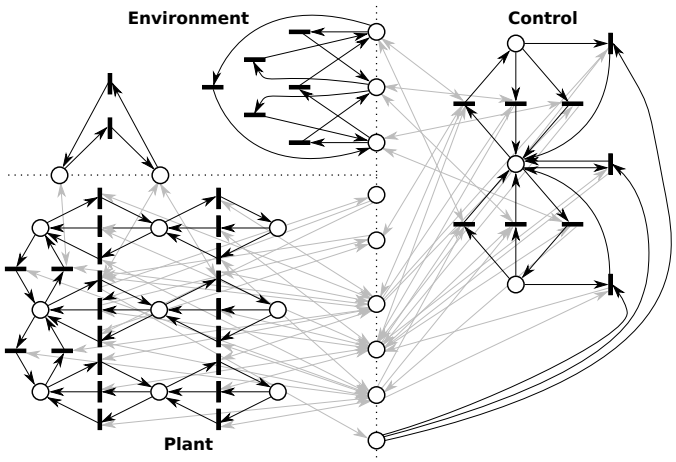


Figure 6.10: Combined Petri Net, automatically generated from the DSL models.

As much as possible is still done automatically though, such as merging input models and metamodels and splitting of their result.

6.1.3 Requirement 3: Process Modelling

All these domain-specific languages, models, and activities are present in the Modelverse. There was, however, no order in which this should happen, and how data has to be utilized. An FTG+PM model can then be added, which links everything together and structures the execution order of activities. This process model, as was shown in Figure 6.1, can not only be used for documentation, but also for enactment.

During enactment, modellers are prompted with activities that they have to perform, meaning that they are often not burdened with intermediate artefacts. For example, the composed Petri Net, shown in Figure 6.10, is never shown to the modellers, though it is generated and is the model that is analyzed. Even better: not a single Petri Net model is shown to modellers at all, as this translation also happens automatically. With the process model, we see that there are only a few manual operations, where domain experts must model their part of the system. Afterwards, they get feedback on whether the total model is safe or not. Modellers remain unaware of the technology that obtains this result.

6.1.4 Requirement 4: Multi-User

As is natural in this case, multiple users will collaborate on this single power window case study. For example, there will be a plant engineer and control engineer, both experts in their respective domain. Given that the process model dictates that all manual modelling operations happen concurrently, up to five different users will work concurrently for this problem only. As the Modelverse is a server, there will be many models being worked on in parallel as well.

To allow for these multiple concurrent users, the Modelverse spawns new tasks for each activity being executed in the process model. Users are then notified of the task they

should connect to, making it extremely simple for users to work concurrently. Inside the Modelverse Kernel, these different concurrent tasks are coordinated using SCCD to offer concurrency and fair sharing of computational resources.

6.1.5 Requirement 5: Multi-Service

While we previously presented activities in different formalisms, we often want to reuse existing tools. For example, Petri Net reachability analysis has been implemented by many specialized tools, and should therefore be easy to reuse. While it is possible to reimplement in the Modelverse, using the procedural action language, this is unlikely to be as efficient.

The Modelverse has built-in support for several external services, implemented in a variety of programming languages. For example, by default there is support for a file creation service (writing out files) and a HUTN compilation service (used when compiling textual models). Additionally, several more complex services are included, such as PythonPDEVS (for DEVS simulation) and LoLA (for Petri Net analysis). For this case study, we have used LoLA to perform the state space analysis, based on the specified safety query.

This single process already used multiple services: the HUTN service to compile the uploaded models and the LoLA service to analyze the generated Petri Net. Note again that in the Modelverse, modellers will be unaware that LoLA is used to do the analysis, as this is completely transparent.

6.1.6 Requirement 6: Multi-Interface

The different modellers and their different background naturally raises the need for multiple interfaces. The simple interface provided by the Modelverse, and the associated client wrapper in SCCD, makes it easy to implement new interfaces. As could be seen, we have implemented four different types of interface: textual, graphical, API, and direct XML/HTTPRequest based. All these interfaces can equally well be used to work on the process model of the Power Window case study.

In the process, each manual operation spawns a new instance of the prototype GUI, although it is trivial to spawn a different GUI. The only requirement is, however, that the GUI is able to interact with the Modelverse in one way or the other.

6.1.7 Requirement 7: Model Sharing

In the Modelverse, everything is a model. This is great in the context of model sharing, as this encompasses literally everything in the Modelverse. Users can share and reuse not only models, but also metamodels, activities, and processes.

In the Power Window case study, the languages are probably created by other people than the modellers in those languages. As such, there is already a need to share different models between one another. Similarly, multiple modellers might want to work on the same model and therefore require access to this exact same model. Sometimes, languages can be reused from other projects, such as the Petri Nets and Encapsulated Petri Nets in the Power Window case study. Indeed, these different languages are general purpose and many applications can be found. By reusing these languages, we are also reusing the activities

defined on them, such as analyzing the reachability graph. For the Power Window case study, this means that we wouldn't have to reimplement almost half of the languages and activities used.

6.1.8 Requirement 8: Access Control

Logically, it should be possible to limit the amount of data being shared, as mandated by access control. For example, in the Power Window case study only plant engineers can be allowed to open the plant model, thereby hiding the intellectual property stored in the model. Similarly, languages can be marked as readable by the engineers, but only language engineers have write permissions, thereby preventing modellers from messing with the language itself. This access control was implemented in the Modelverse by using groups and users, meaning that different users can share a set of permissions.

6.1.9 Requirement 9: Megamodelling

Megamodelling allows users to link different models together, for example through traceability links. In the Power Window case study, two types of megamodels are actually used. The first type is the FTG+PM that is stored, which references the different models, languages, and activities used throughout the process. A second type of traceability links are stored in a separate model that binds other models together. These links were necessary to allow feedback to the modellers when a counter-example is found, but is also required in model transformations to find the target element of a source element.

6.1.10 Requirement 10: Portability

Finally, throughout this thesis many references were made to the Modelverse being portable. This is also reflected in the Power Window case study, where modellers are never exposed to the platform of the Modelverse. It is therefore perfectly possible to implement a different Modelverse, still adhering to our specification, and have it do exactly the same thing. And indeed, as part of a Bachelor's honour project, a new stand-alone implementation of the Modelverse Kernel was constructed which uses JIT compilation. While both implementations were written in Python, a great language for prototyping, this choice was not made to cope with limitations of the foundation.

Additionally, the use of services makes it possible to communicate with existing tools, whatever platform they run on. For example, LoLA, as used in the Power Window case study, is implemented in C, though this could be anything as long as it could be wrapped in one way or the other.

Summary

For each of the original requirements, we illustrated how the Modelverse addressed them, using a single Power Window case study. While this case study was relatively small, it still touched upon all requirements that we originally identified. Full support for this case study thereby indicates that the Modelverse can be considered as fulfilling the requirements we set out for.

6.2 Live Modelling

We first consider the modeller, being the most common type of user. The modeller is responsible to create an accurate model of the system in question. In particular, this means that the behaviour on the model should be identical to the behaviour on the actual system, given some property of interest. As such, the user is mostly concerned with model creation and comprehension. This is where model debugging [294] comes into the picture: the model and the system do not agree on some property, while this should be the case. Model debugging can then be used to find out why they do not agree, and potentially later on solve the issue.

One technique that is know to increase model comprehension, is live modelling [289, 302]. Originally coming from the programming domain, live programming allows programmers to dynamically change the source code of a running application. This increases code comprehension, as there is direct feedback on changes due to the reduced edit-compile-debug cycle. Live modelling then applies this same technique to modelling, where an executable model is modified directly, with the changes being dynamically taken into account in the running execution (e.g., a simulation).

We present an explicitly modelled framework for the definition of live modelling languages: all aspects of the approach, including the process, are explicitly modelled and are part of the contribution. Our approach therefore relies on support for MPM, as otherwise the use of an FTG+PM would not be possible. Additionally, our approach is especially useful in the context of MPM, where various domain-specific formalisms are used and processes can be enacted.

6.2.1 Motivation

Recently the use of models has shifted more and more from documentation to execution, for example through code generation [154] or simulation/interpretation. This brings forth the need for debugging the execution, as seen in the programming community. While there is a growing interest in model verification, not all models can be verified due to the size of the state space, or due to lacking (efficient) tool support. Furthermore, model verification can indeed help find whether a system is correct, but it is often unable to track down the source of the violation. As such, debugging is a vital part of the modelling process.

Commercial modelling and simulation tools often provide limited support for debugging. The research community has recently made several contributions that enable specific debugging features for several types of languages [50, 63, 182, 218, 333]. Compared to code debugging, for which there is a wide variety of debugging operations [342], current modelling tools are still in their infancy in terms of features, applicability, and usability. It is therefore tempting for system developers to debug the automatically generated code directly, instead of the model itself [96].

Live programming [276] is an advanced feature of several programming tools, allowing programmers to modify the code of applications while the application is running, immediately having the new code integrated in the *running* application. There is no apparent compile and re-run cycle, reducing the cognitive gap between code and program. More importantly, the state of the running application is (partially) retained between versions, removing the need to redo all operations up to the point in time where the change was made. While there

are already several tools that support live programming, making a programming language "live" is done ad-hoc, and is refered to as a black art [55]. As such, it is difficult to transpose liveness techniques between languages.

We transpose the essence of live programming to the modelling domain, in a generic way. Contrary to live programming, where only a single language is considered most of the time, domain-specific modelling raises the need for many different domain-specific languages. Many of these domain-specific languages only have a handful of users, rendering the investment for implementing live modelling techniques in an ad-hoc way difficult to justify. Therefore, live modelling should be implemented in a generic way, making it applicable to many modelling languages with minimal effort. Support for live modelling was identified as a key feature to advance the usability of model-driven techniques [168]. The research question thus is "how can live programming concepts be ported to the modelling domain, making them generically applicable". Despite mostly being presented as a debugging operation, live languages can be applied to other situations as well, such as education or model comprehension in general.

To tackle this problem, we deconstruct the traditional live programming process, and reconstruct it in the context of modelling by applying concepts and techniques from live programming to executable modelling languages. The various aspects of liveness are categorized in generic activities and formalism-specific activities. All formalism-specific activities are distilled into a single operation, which we term *sanitization*. To make a new language live, only the sanitization operation should have to be updated, while reusing all other aspects of live modelling. Note that, as liveness only applies to executable modelling languages, we only consider executable modelling languages (or "formalisms").

We distinguish between three types of executable modelling languages for which the implementation of the sanitization operation is fundamentally different. For each, we present a representative example, used throughout as a running example: Finite State Automata (FSAs) [143], Discrete Time Causal Block Diagrams (DTCBDs) [61], and Continuous Time Causal Block Diagrams (CTCBDs) [61].

6.2.2 Background

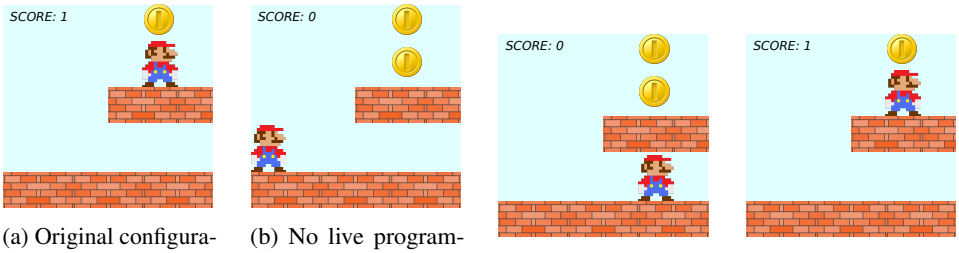
We first present the necessary background in live programming and executable modelling.

Live Programming

Live, or interactive, programming aims to bridge the "gulf of evaluation" [180, 289]. It allows users to update the source code of an application while it is running, with changes being applied instantly in the running application. There is therefore no need to manually recompile, restart, and rerun the program up to the point of execution when the modification was made. This has several advantages, such as decreasing the length of the edit-compile-debug cycle, and offering users immediate insight in the effect of code changes. An example of live programming, as implemented by ElmScript [74], can be seen online[1].

Basically, the process of live programming is as follows.

[1] `http://debug.elm-lang.org/`



(a) Original configuration. (b) No live programming. (c) Recorded event. (d) Real-time.

Figure 6.11: State of the game before and after decreasing the jump height parameter.

1. A developer writes code in a programming language.
2. The (valid) code is compiled to instructions for the specific machine.
3. The instructions are loaded into memory, and storage is allocated for execution.
4. The program is executed, which performs operations on the program and its state.
5. The developer modifies the code of the program, concurrently with execution.
6. The modified code is compiled to new instructions.
7. The program merges its old instructions and state with the new instructions.
8. The program executes the new instructions.

With the exception of the 7th item, these steps are identical to the workflow of normal programming. Normally, however, the new instructions are only executed in a new invocation of the program. The merge operation, therefore, is the only new operation in live programming (from a functional point of view). The merge operation alters a running program to incorporate changes unknown at compilation time, by merging the updated set of instructions with the old state of the running program. Specifically, new instructions that do not have an execution context are merged with old instructions and their associated execution state. As data is also merged, such as the value of variables, information from the old program must be combined with the new instructions.

Data merging is intentionally left vague, as many approaches exist. Three categories were proposed [194], depending on how much data is copied: no live programming, recorded event, and real-time. We illustrate all three with a game example, similar to the ElmScript example. The game is a simple platform game, where the jump height of the character is updated during execution. The game's current state is shown in Figure 6.11a, where the character jumped onto the platform and, in the meantime, collected one coin. If the character were to jump, the coin is collected and the score is increased to 2.

No live programming is the most basic, where no information is passed between executions. Upon recompilation, the currently running application is terminated and restarted afresh. This approach does not implement live programming at all, and can easily be replicated without any modification to the programming language itself. All that is required is an automatic restart of the application after a change is detected. In the game example, the character is respawned at the beginning and the score is initialized to zero. This is shown

in Figure 6.11b, where the character has respawned and all coins have been reset as well. From this point onwards, the jump height is reduced and the character will be unable to jump on the platform. In conclusion, no state is retained.

Recorded event takes over the history of all inputs sent to the old running application. The new program is then executed with these simulated events, making it seem as if the inputs sent to the old program were sent to the new program. This approach is used in programming languages such as ElmScript [74]. For performance reasons, the program is often not completely re-executed, but only dependent functions are re-evaluated. In the game example, our character might switch location and score, depending on what these values would be if the exact same inputs were given in the new application. When the jump height parameter is decreased, we suddenly find the character below the platform, instead of on top of it. This is shown in Figure 6.11c, where we see the character below the platform: the jump we did before did not reach the same height, which made the character unable to reach the platform. Subsequent actions, such as moving to the right, were still replicated, but in a different context: below instead of on top of the platform. In conclusion, the input history part of the state is retained.

Real-time takes over the complete history of the old running program, but merges in new instructions to be used in the future. The new program is effectively a rewritten version of the old program, which just continues computation. This approach is used in programming languages such as Smalltalk [123], and is often also termed *fix and continue*. In the game example, our character will be at the same location and have the same score as before, but changes will take effect from that point onwards. When the jump height parameter is decreased, we find it impossible to jump as high as we could before, though our current location remains unchanged. This is shown in Figure 6.11d, where we see no immediate change. From this point onwards, however, we are unable to get the coin right above us, as the character can no longer jump that high. In conclusion, the complete state is retained.

Executable Modelling

Modelling has historically mostly been used in the form of documentation of a separate coded application. Recently, however, executable modelling has gained popularity, where the model itself becomes the final application, without additional coding effort. In this case, the model is not necessarily used as documentation or to generate skeleton code, but its execution becomes detached from programming. In essence, models have gained semantics, for which two main categories exist, as presented before: denotational and operational semantics.

Operational semantics has to store extra execution data, and therefore often makes the distinction between a design and runtime metamodel. The design metamodel is the metamodel that is used by the designer when creating the model. It has all the necessary constructs for design, but is not concerned with the execution. For example, in Finite State Automata (FSAs), a State only has a *name* and *initial* attribute. The runtime metamodel, however, has additional information required for execution. For example, the state now still has its *name* and *initial* attribute, but additionally has a *current* attribute. This attribute stores a boolean containing whether or not this is the current state of the execution. While this information is required for the execution, as it needs to be stored somewhere, it is invisible to the designer.

Due to the distinction between these two types of metamodels, multiple models are actually required for operational semantics: the design model is first translated to a runtime model, thereby initializing it (e.g., setting the *current* attribute to the *initial* attribute). The actual operational semantics is subsequently executed on this intermediate runtime model.

6.2.3 Running Examples

We will use several running examples throughout this section. For all formalisms, we present a simple model on which we use live modelling. Modelling languages can have widely varying semantics, including non-determinism, event-driven behaviour, timing, etc. While implementing live modelling techniques for each of these categories will be different, one essential difference that has an effect on live modelling is the types of changes that can be made. We identify three types of languages: two that gain semantics through operational semantics (i.e., they manage the state themselves), with support for breaking and non-breaking changes, and one that gains semantics through denotational semantics (i.e., they delegate execution and states to another formalism). The distinction between breaking and non-breaking changes stems from the language evolution community [197], where changes to the metamodel can be considered to break the instances. With breaking changes, the instances have to be adapted, for example when adding a new mandatory attribute to a class. This can be either resolvable (e.g., when the attribute has a default value) or non-resolvable (e.g., when the attribute has no default value). With non-breaking changes, the instances do not have to be adapted, for example when adding an optional attribute to a class.

With *operational semantics and breaking changes*, conforming changes on the design model might result in non-conforming changes on the runtime model. For example, in live programming, the currently executing line of code can be removed. In this case, the change in the design model (source code) is a valid piece of program code, but when this same change is mapped to the runtime model, the current instruction pointer is also removed, making the runtime model invalid. To solve inconsistent states after such a change, a new line of code must be selected as the currently executing line of code, which can often not be done automatically. Note that it is not possible to switch the instruction pointer to a different line of code, as we do not have access to the state values themselves. As a representative example of a formalism with breaking changes, we choose Finite State Automata.

With *operational semantics and non-breaking changes*, conforming changes on the design model always result in conforming changes on the runtime model. For example, in live programming, variables can be added or removed, and their values cannot be changed. In this case, the change in the design model (source code) is a valid piece of program code, and when the same changes are mapped to the runtime model, the runtime model stays valid. Note that it is not possible to change the values of variables themselves, as we only have access to the source code, not to the execution information. As a representative example of a formalism with non-breaking changes, we choose Discrete Time Causal Block Diagrams.

With *denotational semantics*, execution is delegated to another formalism. For example, in live programming, the codebase is first translated to another programming language, for which live programming is supported. To solve the inconsistent state, the sanitization process executes at the target language level, thereby reusing existing sanitization approaches.

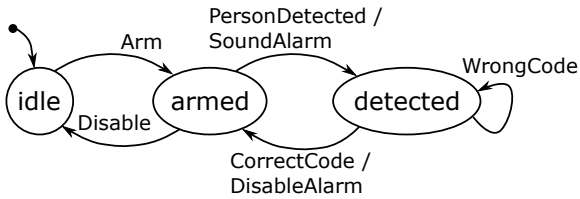


Figure 6.12: Example FSA of a home security alarm system.

In the end, however, the modeller is likely only an expert in the source language, and might even be unaware of the existence of a target language. As a representative example of a formalism with denotational semantics, we choose Continuous Time Causal Block Diagrams, which we map to Discrete Time Causal Block Diagrams through discretization and optimization.

For readability, we present our approach using these three different formalisms, all introduced in Chapter 2. Many languages support different types of changes, such as some that are breaking (*e.g.*, the executing line of code) and others that are non-breaking (*e.g.*, variable values). Therefore, a composite merge rule is often required, which handles all aspects simultaneously. We explain all used formalisms next, along with an example model.

These three types of languages are exhaustive: a change is either breaking (i.e., requires changes) or non-breaking (i.e., doesn't require changes). For breaking changes, two resolution methods exist, both of which are handled. Denotational semantics does not consider the difference between breaking and non-breaking changes, as it merely relies on the underlying semantics for this. As such, denotational semantics is only considered in combination with one type of changes.

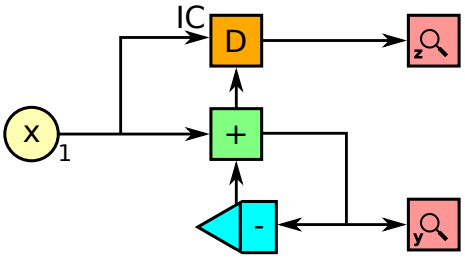
Finite State Automata

The Finite State Automata (FSA) language [143] is a modelling language used to model reactive systems with discrete state, and was previously mentioned in the Background chapter. Our running example is the same, specifically that of a home security alarm system, shown in Figure 6.12.

The FSA language is an example of a language with potential breaking changes: the only state of the model is the current state, which is explicitly present and can thus be removed. If the user deletes the current state, execution can only resume when another state is chosen as the current state. This can be resolved either manually (breaking, non-resolvable) or automatically (breaking, resolvable).

Discrete Time Causal-Block Diagrams

The Discrete Time Causal Block Diagrams (DTCBD) language [61] is a modelling language used to model a complex system of equations, and was previously mentioned in the Background chapter. Our running example is the same, specifically that of the equation $y = x - y$, shown in Figure 6.13a. The equivalent set of equations is shown next to it.

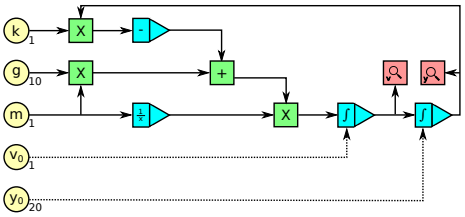


(a) Example DTCBD, containing an algebraic loop.

$$\begin{cases} y(t) = x(t) - y(t) \\ z(t) = \begin{cases} x(t) & \text{if } t = 0 \\ y(t-1) & \text{if } t > 0 \end{cases} \end{cases}$$

(b) The equations represented by the example DTCBD model.

Figure 6.13: Example DTCBD.



(a) Example CTCBD model.

$$\begin{cases} v(t) = \int_0^t \frac{m \cdot g - k \cdot y(t)}{m} dt \\ y(t) = \int_0^t v(t) dt \end{cases}$$

(b) The equations represented by the example CTCBD model.

Figure 6.14: Example CTCBD.

The DTCBD language is an example of a language with exclusively non-breaking changes: delay blocks have a memory of their previous iteration, but this is no longer necessary when the block is deleted. The runtime state of the model is an aggregation of the memory values, which the user cannot manipulate directly. When a delay block is added, the state needs to be updated accordingly by initializing a new state variable. Similarly, when a delay block is removed, part of the state is removed. It is impossible for a conforming design model to result in a non-conforming runtime model.

Continuous Time Causal-Block Diagrams

The Continuous Time Causal Block Diagrams (CTCBD) language [61] is an extension to DTCBDs, introducing two continuous blocks: an integrator and derivator. Our running example is the same, specifically that of a mass suspended on a string, shown in Figure 6.14a. The equivalent set of equations is shown next to it.

The CTCBD language is an example of a language with denotational semantics: to execute the model, it is first translated to an equivalent DTCBD (with respect to some properties), which is then executed instead. It does not matter whether the target language has breaking or non-breaking changes, or has denotational semantics itself, as live modelling is assumed to be supported for that language already. As such, we build on top of the live modelling functionality that was developed for our other running example.

While we acknowledge that DTCBDs and CTCDBDs look similar, there is a non-trivial n-to-n mapping between both languages due to the discretization and optimization. Even

though many concepts can be reused between the two, the mapping exhibits most of the complexities normally associated with traceability links in denotational semantics.

6.2.4 Approach

We start our approach to live modelling by deconstructing the current process for live programming schematically, and then generalize the concepts and processes to modelling. This results in a general framework for live modelling, that can be applied to any (domain-specific) executable modelling language. Programming languages also fit this framework, as they can themselves be seen as an executable modelling language.

Deconstructing Live Programming

The first step in our work is the deconstruction of the live programming process. This process consists of artefacts (*i.e.*, files or structures in memory) and modifications (*i.e.*, operations on these artefacts). An overview is shown in Figure 6.15.

Artefacts We distinguish three artefacts: code, instructions, and the running program.

The *code* is the textual notation that represents a program, created by the developer. Code is often persisted as a text file. It is the only artefact programmers should edit; they should not edit any subsequent (automatically generated) artefacts. An example is a *C++* source code file.

The *instructions* are the result of compiling the code. Consisting of a set of instructions and data, which can be interpreted by the machine. Execution-time concepts are not yet considered: variables have no value, nor is there a currently executing line of code. The compiled program is only an "intermediate" form: it is an optimized version of the original code, and is easier to read for a computer. As part of the compilation process, the program is instrumented with extra information, such as mapping variables to registers. An example is a compiled *C++* program in *ELF* format. It is important to note that these instructions are semantically equivalent to the original code.

The *running program* is the actual program loaded in memory, including its state. It is executed by the machine and is very similar to the compiled program, but includes runtime information (the state). Multiple versions of the same program can execute at the same time without sharing state (*i.e.*, memory): each program runs independently of the others. Even when the instructions are changed (*i.e.*, in self-modifying code), these changes only take effect on the running instance. Thus, program execution can be defined as the continuous updating of the artefact itself. An example is the memory used for executing an *ELF* file, encompassing both the instructions and the execution data.

Operations We distinguish five operations between these artefacts: compilation, initialization, execution, modification, and merging.

Compilation (code to instructions) transforms a human-readable piece of code to a machine-readable representation. This process involves steps such as making implementation decisions and register allocation. The generated machine code remains semantically equivalent to the original code.

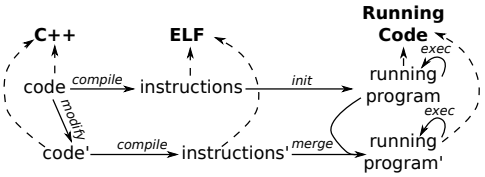


Figure 6.15: Diagrammatic overview of live programming. Full lines represent operations, dotted lines represent typing relations.

Initialization (instructions to running program) loads a compiled program into memory and initializes its state at the start of execution. Apart from initializing the state, the machine code is copied to memory.

Execution (modification of running program) modifies the program by changing the data, or by changing the instructions (self-modifying programs). Execution typically only alters the state of the variables contained in the program.

Modification (modification of code) represents the changes a user makes to the original source code artefact. Arbitrary changes are supported, as long as the result is still a valid instance of the original language (*i.e.*, it can be compiled).

Merging (instructions and running program to a running program) merges the state of a running program with an updated set of instructions. The merge operation is specific to live programming: the currently executed program is merged with the updated instructions. Afterwards, the "new" program resumes execution where the "old" program left off, thereby replacing it. This can be seen as a generalization of the initialization operation: as part of the merge, the state is initialized for new instructions, while it is modified if instructions are removed or updated. We therefore consider initialization a merge with an "empty program".

The live programming process is shown in Figure 6.15, where we explicitly mention the type of artefacts for a specific scenario. That way, the signature of the operations becomes apparent. While live programming environments often offer additional features for performance reasons, such as incremental compilation, these are not functionally required.

Transposition to Modelling

Taking this diagrammatic process, we generalize to the domain of modelling. We port these concepts to the modelling domain: instead of using programming languages and execution on actual machines, we make it platform-independent. Whereas we used a language such as *C++* before, we now assume the artefacts as instances of an executable language. Our approach is a generalization: it can also be applied to programming languages, since they can be seen as executable modelling languages. Their syntax is defined in the language's grammar (cf. metamodel), while their semantics is defined by their mapping onto machine code.

Artefacts First, we transpose the artefacts, which gives us three kinds of models: the design model (code), partial runtime model (instructions), and full runtime model (running

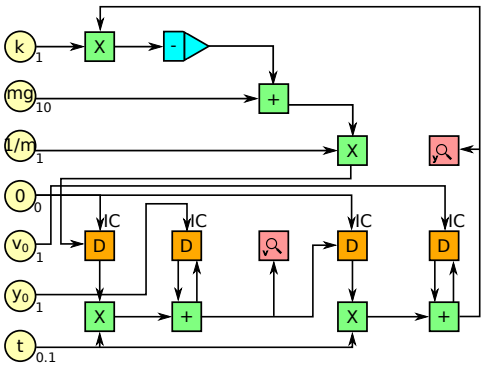


Figure 6.16: The partial runtime model of the example CTCBD, as an instance of the DTCBD language.

program).

The *Design Model* is the equivalent of the *code*. Similar to code, it is the only artefact that the user can edit, and thus also the one that is seen as the "master" copy of the program. Our previous examples of an FSA, DTCBD, and CTCBD model, presented in Figure 6.12, Figure 6.13a, and Figure 6.14a, respectively, are expressed in the design language.

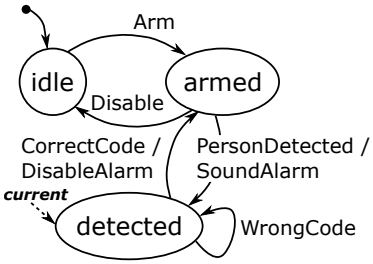
The *Partial Runtime Model* is the equivalent of the *instructions*. Similar to instructions, it has the same meaning as the design model, though it might be pre-processed. If operational semantics is defined for this formalism directly, it can be seen as a retyping operation. In general, however, the structure of both languages might vary significantly (as was the case with *C++* and *ELF*). In the FSA and DTCBD languages, the partial runtime models are equivalent to the design models, since both languages have operational semantics. In the CTCBD language, the partial runtime model differs, as it is a model in the target language: DTCBDs. Figure 6.16 presents a discretized version of the original CTCBD model, in the DTCBD language.

The *(Full) Runtime Model* is the equivalent of the *running program*. Similar to the running program, the full runtime model is a copy of the partial runtime model, extended with additional elements representing the execution state. In Figure 6.17, the full runtime models of the running examples are shown.

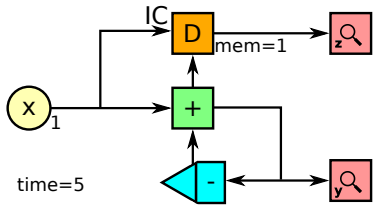
For FSAs (Figure 6.17a), a pointer to the *current state* is added. In the figure, the model is currently in the *detected* state. For execution, the model is updated by changing the current state based on the input events received from the environment.

For DTCBDs (Figure 6.17b), more runtime information is added, as they have a notion of time, represented by the number of iterations. The *time* is incremented each time an iteration is executed. Each iteration, the signal values are (re)computed based on the new input values. For most blocks, their output signal value only depends on their current input values and hence they are stateless. One exception is the delay block, whose output value depends on its input value in the previous iteration. A *mem* runtime variable keeps track of this value, which must be initialized as well.

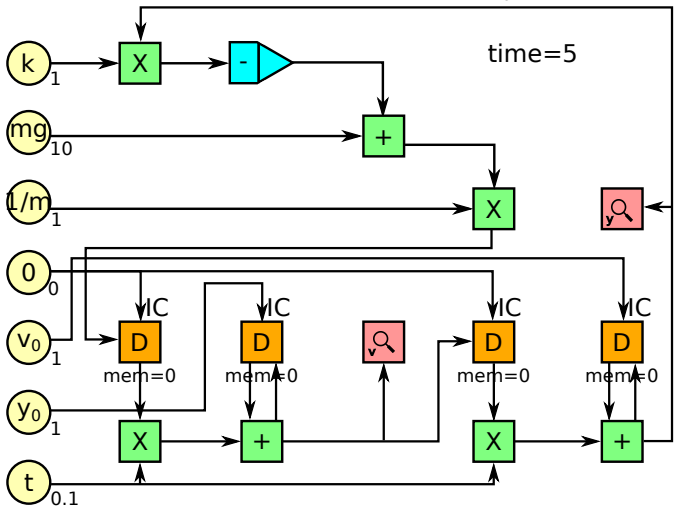
For CTCBDs (Figure 6.17c), the situation is identical to DTCBDs now, as the model was



(a) The full runtime model of the example FSA, during execution.



(b) The full runtime model of the example DTCBD, during execution.



(c) The full runtime model of the example CTCBD, during execution.

Figure 6.17: The full runtime models of the examples.

effectively translated to the DTCBD domain.

Operations Second, we transpose the various operations on these artefacts: retyping (compilation), simulation (execution), modification (modification), and sanitization (initialization and merging).

The *Retype* operation is the equivalent of the *compile* operation. Similar to compilation, it creates a semantically equivalent copy of a model, while retyping it to a runtime model. It does not necessarily have to be a trivial retyping, as potentially the design and partial runtime model have a slightly different structure (*e.g.*, flattening hierarchy). Retyping is thus also responsible for making this translation. As explained before, the partial runtime models for both the FSA and DTCBD language do not contain additional information. The retyping operation is therefore trivial in this case. For CTCBDs, the retyping actually casts the model to a language for which semantics exists. This operation involves discretization (one CTCBD element is mapped to multiple DTCBD elements) and optimization (multiple CTCBD elements are mapped to one DTCBD element). After this discretization, however, the case becomes identical to DTCBDs for the remainder of the live modelling process.

In all cases, traceability links are created between the various elements to help in future operations. For example, in the FSA, the design state is linked to the equivalent partial runtime state, such that on subsequent operations, it is known that this state has already been converted before, and therefore does not need to be recreated again.

The *Simulation* operation is the equivalent of the *execution* operation. Simulation computes the next state of the full runtime model and updates it in-place. For the FSA language, the next state of the model is computed by processing an event from the environment, and executing an enabled transition by changing the current state and (optionally) raising output events to the environment. For the DTCBD and CTCBD languages, there is no external input or output. The next state of the model is computed by, for each block, computing the output signal value based on its input values. This requires detecting loops and solving them if they represent a set of linear equations. For delay blocks, the output value is equal to its value in memory (or the initial condition at the first iteration when the memory value has not been set yet). The memory value is overwritten by the current input value of the delay block. At the end of computing the next value of all blocks' output signal values, the iteration counter is incremented. As we are operating on models, and not on generated code, we do not need to consider the technical aspects of replacing executing code: the model is updated in-place and the simulation algorithm picks up these changes in the next step. Note, however, that the simulation algorithm does *not* take care of initialization, as is usually the case. Indeed, normally the first step of simulation is to initialize variables, which is now unnecessary: all information is stored and read out from the model itself. Some parts of the simulation algorithm still need to be done, which are not related to initialization of the model, but initialization of the simulation algorithm, such as topological sorting for DTCBDs.

The *Modification* operation is the equivalent of the *modification* operation in programming. Similar to modification in the programming domain, users can only modify the design model. Since all other artefacts are automatically generated, the design model is the only artefact they are familiar with. While the user never edits the partial or full runtime models directly, the design model can be freely modified. As usual, Create-Read-Update-Delete (CRUD) operations are supported on the model. This boils down to Creating new elements and attributes, Updating the values of attributes, and Deleting elements and attributes. Note that reading does not modify the model, and is therefore ignored.

To highlight the various types of changes, each language has a different type of change. For the FSA language, users can change the triggers on transitions, remove transitions, create new states, and so on. A modified FSA design model is shown in Figure 6.18a, where the *detected* state is removed (Delete). For the DTCBD language, users can instantiate new blocks, delete existing blocks, add or remove dependencies, and so on. A modified DTCBD design model is shown in Figure 6.19a, where the value of $y(t)$ is multiplied by two, thereby changing the algebraic loop (Create). For the CTCBD language, users can instantiate new blocks (including the integrator and derivator), delete existing blocks, add or remove dependencies, and so on. A modified CTCBD design model is shown in Figure 6.20, where the gravitational constant is altered (Update). For all languages, the design models must conform after the modifications. Note that different types of operations were applied for each formalism: removing a structural element in FSAs, creating several structural elements in DTCBDs, and changing a parameter in CTCBDs. This highlights the various types of operations that can be done on the design model, all of which are reflected

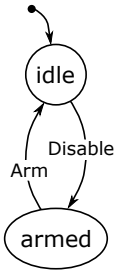
in the running simulation.

The *Sanitization* operation is the equivalent of the *merge* operation. While it is indeed a merge operation, it was renamed to sanitization to prevent confusion with the existing term model merging [54]. The operation creates a full runtime model from a (new) partial runtime model and an (old) full runtime model. As the sanitization is domain-specific, it is difficult to make general claims about this operation: it is whatever the language engineer wants it to be. Nonetheless, the sanitization function can be sure that both input models will conform to their metamodel (which the language engineer can define), and must ensure its output conforms to the full runtime metamodel. Sanitization includes initialization (where the runtime state is empty) and the live modelling "merge", where the runtime state is taken into account. As discussed previously, sanitization is fundamental to live modelling support, and as such, it is discussed in detail next. As was the case with the retyping operation, sanitization makes use of traceability links, linking elements from the partial runtime to the full runtime. Traceability links are used to correctly migrate the state of the full runtime model to the right elements in the partial runtime model. For example, in the FSA, a state in the partial runtime model without any traceability link is considered to be a new element, while a state in the full runtime model without a traceability link is considered to be removed, possibly triggering a problematic situation when this was the current state of the simulation.

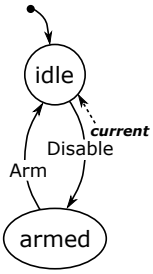
Sanitization The sanitization operation is largely dependent on the types of changes to be merged (*i.e.*, breaking or non-breaking), but remains a language-specific operation. Therefore, a manually defined version needs to be created for each new language. Nonetheless, our decomposition has shown that this is the only operation that needs to be added, in order to provide live modelling for that formalism. Depending on how the sanitization operation is implemented, any of the three types of live modelling (*i.e.*, none, recorded event, or realtime) can be implemented. We leave open the medium in which this operation is expressed (*e.g.*, procedurally using code or declaratively using model transformations). The presented code snippets therefore do not restrict sanitization to a procedural approach. We present the sanitization operations for both types of state, using our running example: the FSA, DTCBD, and CTCBD formalisms. For all three, we present realtime live modelling. Note that, similar to live programming, sanitization can only happen when the state is consistent (i.e., inbetween two execution steps).

Breaking Changes When breaking changes are possible, the runtime model might have to be made conforming to its metamodel again. For example, when users remove the current state in the design model, the equivalent state in the runtime model also has to be removed, thereby violating the constraint that the runtime model has exactly one current state. In that case, a new state of the updated running system must be defined. Changes to any other aspect of the design model are irrelevant to the running system, and are just taken over.

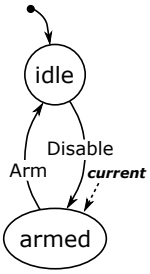
Resolving this breaking change is the core task of the sanitization operation. There are three options: reset the current state to the initial state (automated, so resolvable), prompt the user for a new state (manual, so non-resolvable), or disallow the change completely (disallow breaking changes). For the new design model in Figure 6.18a and the old full runtime model in Figure 6.17a, the first two options are presented. Figure 6.18b shows



(a) Modified design model.



(b) Automatic resolution.



(c) Manual resolution to "armed" state.

Figure 6.18: Sanitization in FSAs.

automatic resolution where, in this case, the system chooses the default state (the "idle" state) as the new current state. Figure 6.18c shows manual resolution, where the user chooses the "armed" state as the new current state. Figure 6 shows the pseudocode of the sanitize operation for FSAs.

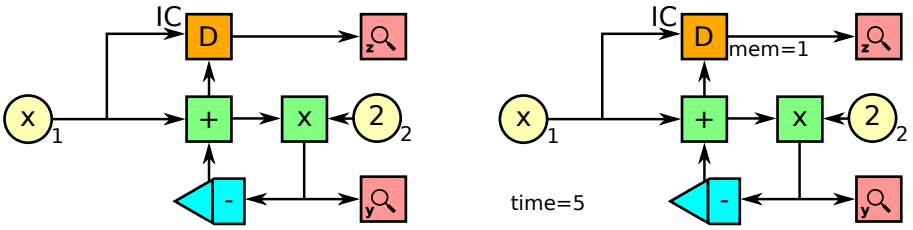
ALGORITHM 6: The FSA sanitize operation.

```
if isInitialized() then
    currState ← getCurrentState(M_F^old)
    if not currState ∈ M_P^new then
        if automaticResolution then
            currState ← getInitialState(M_P^new)
        else
            if disallowChange then
                raise Exception
            else
                currState ← userChoice(M_P^new)
            end if
        end if
    end if
else
    currState ← initializeState(M_P^new)
end if
```

Changes resulting in an undefined current state could also be explicitly disallowed. We did not pursue the direction of disallowing design model changes, as we explicitly want all modifications to be possible.

Non-Breaking Changes For non-breaking changes, any change the user makes always reflects on a conforming runtime model. In contrast to breaking changes, where resolution is required, non-breaking changes don't require significant changes to the runtime model.

In our example DTCBD language, only operations on the integrator, derivator, and delay blocks have any influence. Since each block and connection has its own *signal* and *memory*,



(a) New design model.

(b) Sanitized runtime model.

Figure 6.19: Sanitization in DTCBDs.

removing a block or connection only affects that specific signal. In further simulation steps, however, the change will of course have its effects on other elements as well, as it propagates through the system. It is possible, however, to add new parts to the state (*i.e.*, add new blocks or connections) or remove parts of the state.

ALGORITHM 7: The DTCBD sanitize operation.

for all $block \in M_P^{new}$ **do**
  if $block \in M_F^{old}$ **then**
    $oldSignal \leftarrow getSignal(M_F^{old}, block)$
    $setSignal(M_P^{new}, block, oldSignal)$
  else
    $initializeSignal(M_P^{new}, block)$
  end if
end for
if $isInitialized()$ **then**
  $iterations \leftarrow getNumberOfIterations(M_F^{old})$
  $setNumberOfIterations(M_P^{new}, iterations)$
else
  $initializeNumberOfIterations(M_P^{new})$
end if

When sanitizing, we take the structure from the partial runtime model, which we augment with the runtime data from the full runtime model. In the case of DTCBDs, the runtime information consists of (1) the current simulation time; and (2) the memory of delay blocks, derivators, and integrators. Blocks that were not present in the full runtime model are initialized as usual, since they are new. Blocks that were present, however, have their state copied from the full runtime model. The pseudocode of the sanitize operation for DTCBDs is shown in Algorithm 7.

An example of sanitization is shown in Figure 6.19. In this figure, we see the new design model in Figure 6.19a, and the resulting full runtime model in Figure 6.19b. The full runtime model consists of the structure of the partial runtime model, combined with the values of the old full runtime model. In this case, the value of the *t* variable (representing the current iteration of the simulation), as well as the memory value of the delay block, are copied.

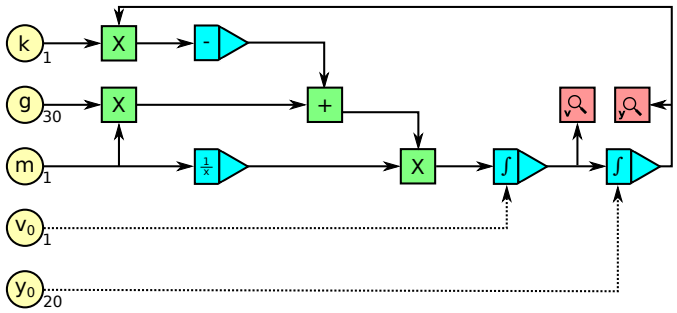


Figure 6.20: New design model for CTCBDs.

Denotational Semantics For denotational semantics, the sanitization is done at the level of the target language, and will therefore be any of the previously mentioned approaches. Sanitization might require traceability information to be present. This information links the various models to be merged together, as indeed the source and target partial runtime models can vary significantly. Using traceability links, elements in different languages can be connected to their equivalent counterparts. Some elements, such as the integrator in CTCBDs, will have traceability links to multiple elements in the DTCBD partial runtime model, as it was expanded (1-to-n mapping). Other elements, such as a constant block in CTCBDs, might have traceability links to a shared element in the DTCBD partial runtime model, as it was partially optimized away (n-to-1 mapping).

The sanitization process is completely identical to that of operational semantics in terms of traceability information: information is stored during retyping and sanitization, and is subsequently used in the next sanitization phase to identify equivalent elements. The only difference is that there is no longer a 1-to-1 mapping, but an n-to-n mapping. Nonetheless, during all phases of live modelling, traceability links are still created. Using this information, it is still possible to find out which design element(s) was the source of the current element in the full runtime model. As for each element in the full runtime model the design element is known, it is possible to find out which elements are identical and should have their state copied.

No new sanitize operation is presented, as the DTCBD sanitization operation is reused.

Live Modelling Process

An overview of the approach, for each case, is shown in Figure 6.21 for FSAs, in Figure 6.22 for DTCBDs, and in Figure 6.23 for CTCBDs.

More generally, Figure 6.24 shows an FTG+PM [186] model describing both the different formalisms and processes of live modelling for any executable modelling language. The left side shows the Formalism Transformation Graph (FTG), describing the different formalisms and the transformations between them. The right side shows the Process Model (PM), describing the sequence of operations done by the user and the data dependencies. It includes the artefacts, how they are related, and the process describing the (automatic or manual) operations. The sanitize operation has a dual colour: it is mostly automatic, though it can be manual for non-resolvable breaking changes, where the user is prompted. In the PM, simulation and modification run concurrently: modifications can be made throughout

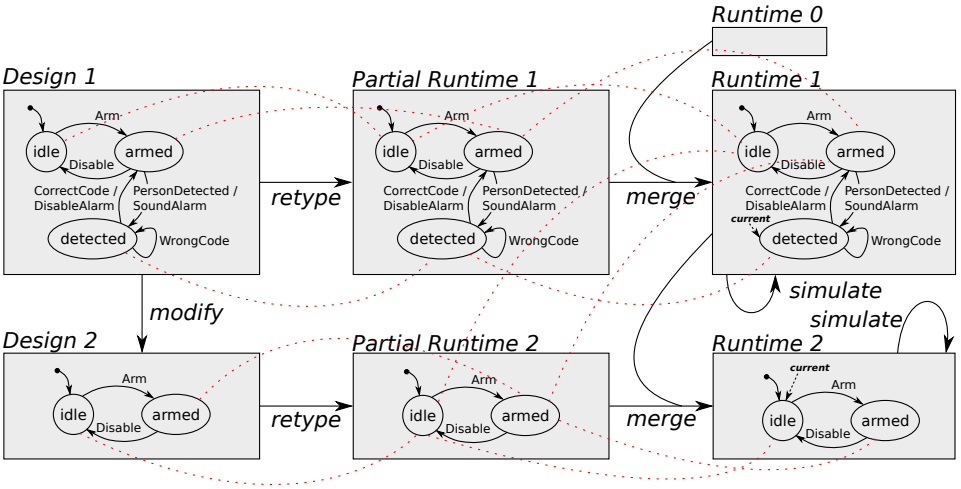


Figure 6.21: Overview of our approach applied to an FSA mode, including traceability links.

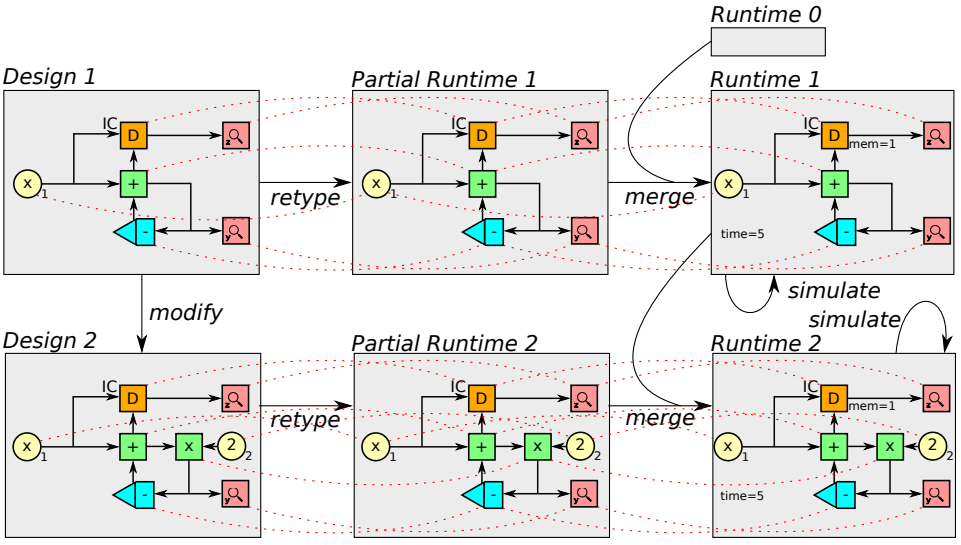


Figure 6.22: Overview of our approach applied to a DTCBD model, including traceability links.

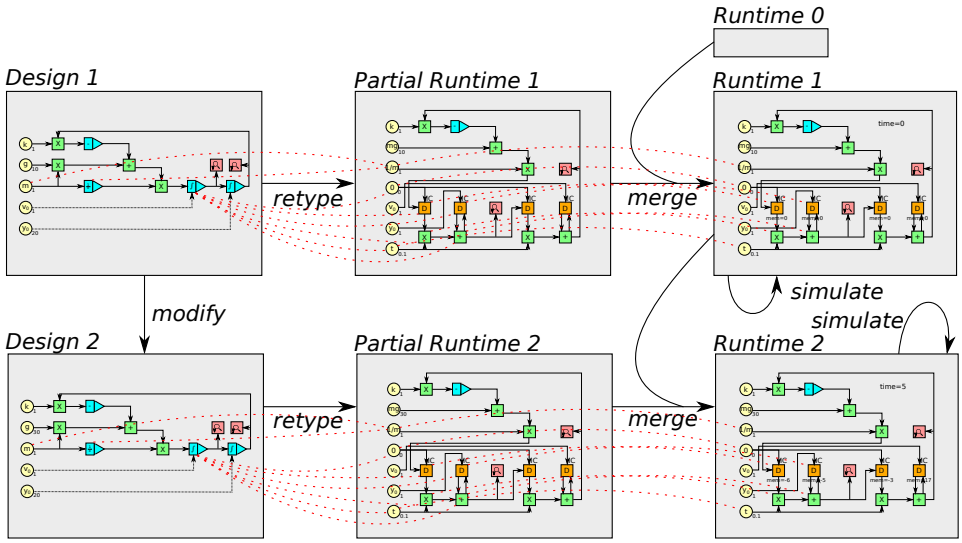


Figure 6.23: Overview of our approach applied to a CTCBD model. Only some interesting traceability links are shown.

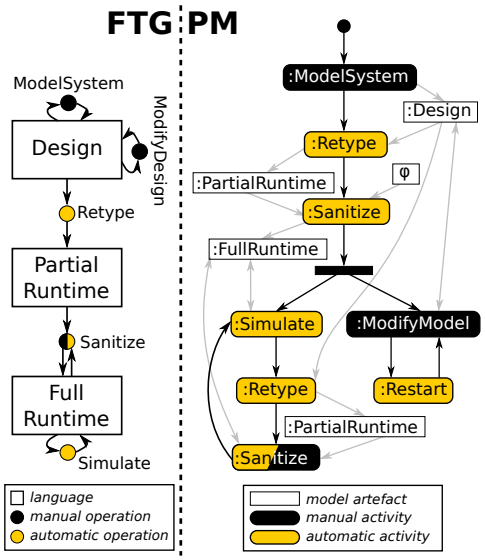


Figure 6.24: Overview of our approach, as an FTG+PM model.

simulation. This is typical for live modelling, in contrast to the mostly linear development process of a single model in ordinary modelling.

Relation to Multi-Paradigm Modelling

Our approach can be considered a Multi-Paradigm Modelling (MPM) approach to live modelling for several reasons.

On the one hand, this approach builds on MPM, as it requires all techniques that are present in MPM: language engineering (e.g., for domain-specific modelling), activities (e.g., for model transformations), and processes (e.g., for enactment). Language engineering is required for the various languages that are used by the approach: design metamodel, partial runtime metamodel, and the full runtime metamodel. All these languages must be created within the tool and should have support for maintaining them. Activities are required to relate the various languages and models together, thereby automatically applying the approach. Activities can be implemented in different ways, such as through declarative model transformations or a procedural action language, and are executed to translate between the various models. Processes are required to structure the approach, thereby preventing it from being ad-hoc as the majority of other approaches to liveness. With support for enactment, it even becomes possible to automatically perform the complete live modelling approach. In conclusion, all relevant aspects of the approach are modelled explicitly, as proposed by MPM.

On the other hand, this approach is desirable in an MPM context, as MPM requires the use of the most appropriate formalism(s) for a problem. The most appropriate formalism, however, is likely to be domain-specific and have a rather limited application domain. As such, the number of users of these languages is small, making it hard to justify the effort ordinarily required to make languages live. With the proposed generic approach, languages can more easily be made live with the addition of a "sanitize" operation, significantly lowering the threshold to live modelling and increasing the usability of the language. Increasing the usability of a formalims naturally makes the language more appropriate for its use, thereby strengthening the MPM approach.

6.2.5 Evaluation

To assess the feasibility of our approach, we implemented live modelling for the three running examples. Our prototype consists of a single visual modelling and simulation front-end, in which multiple languages can be loaded, including FSAs, DTCBDs, and CTCBDs. This front-end is unaware of live modelling. All operations are defined in the Modelverse [304], our Multi-Paradigm Modelling (MPM) tool. The Modelverse implements all aspects of MPM [316], making it possible to use all aspects of language engineering, model transformations, and process modelling, as required by our approach.

In our prototype tool, users start the live modelling process relevant to the language they want to use. The process can be parameterized with an input model, which is the initial model. If no input model is provided, users start from an empty model. Independent of the initial model, simulation is always started anew, as only the design models are stored. Enactment completely resembles the usual modelling interface, but instead of only having a modelling window, a simulation window is now also present. This simulation window is merely an external program that visualizes the simulation results obtained.

Even during modelling, simulation is progressing, and users will see that the simulation window is updated in real-time. Changes made by the user are not immediately committed to the actual design model, as users might want to group a set of operations together into a transaction. As soon as users are satisfied with the design model, and wish to propagate the changes to the running simulation, they commit the design model. In our prototype implementation, committing can be done by closing the modelling window. When the

window is closed, the manual "edit" activity is finished, and the process enactment continues by stopping the current execution and performing the required translations. Once these are completed, simulation is resumed and a new modelling window is opened with the current version of the design model. Users will immediately see that their simulation is resumed, but now taking into account the new model.

For all the three examples presented below, the exact same tool is used, with the exact same (parameterized) FTG+PM model. Apart from the usual operations that have to be implemented for any executable modelling language (i.e., runtime metamodels, operational semantics, denotational semantics), only the sanitize operation is new, and had to be defined for each language individually. As for the visual interfaces, these are untouched when implementing live modelling, as everything is based on process enactment.

Finite State Automata

The implementation of our FSA live modelling environment is shown in Figure 6.25. To the left, the modelling window is shown, which contains a visual representation of the design model. To the right, the simulation window is shown, which is continuously updated with results from the running simulation. The trace shows the current state throughout time. Although FSAs are untimed, input events can be raised by the user through the simulation interface. The state of the system is constant in between such events; the time plotted on the x-axis is wall-clock time. The FSA model itself is oblivious of the current time.

During exection, the current state ("idle") is removed and the new initial state is set to "armed". Upon committing this change, the model and trace is updated as shown in Figure 6.26. It is shown that, upon making that change, sanitization sets the new current state to the new initial state, which is "armed" in this case. Note that there will always be a single initial state, as this is part of the constraints imposed by the metamodel. The history of the simulation is left as-is, since the history is not rewritten with realtime live modelling. Nonetheless, the current state has no effect on the result of sanitization.

Discrete Time Causal Block Diagrams

The implementation of our DTCBD live modelling environment is shown in Figure 6.27. It is similar to the FSA live modelling environment, as they reuse a lot from the Modelverse and our generic approach. Actually, the only difference related to live modelling is the sanitize operation. Of course, the formalisms also differ, just like the simulation viewer, though these are all independent of live modelling, and would be required anyway, even without live modelling. In the simulation view, probed signals are plotted. It is possible for signals to appear or disappear throughout simulation, when a probe block is added or removed during simulation. This is a design consideration of the simulation viewer if it wants to support live modelling.

During execution, the algebraic loop is resolved and sets both y and z to $\frac{1}{2}$. After some time, the algebraic loop in the DTCBD model is extended with an additional multiplication block and constant 2. The value for z now becomes the output of the addition block, while the value for y becomes the result of the multiplication block. After all elements are connected and changes are committed, the trace is updated, as shown in Figure 6.28. Again, the algebraic loop is solved transparently to the user, resulting in a y value of $\frac{2}{3}$ and a z value of $\frac{1}{3}$.

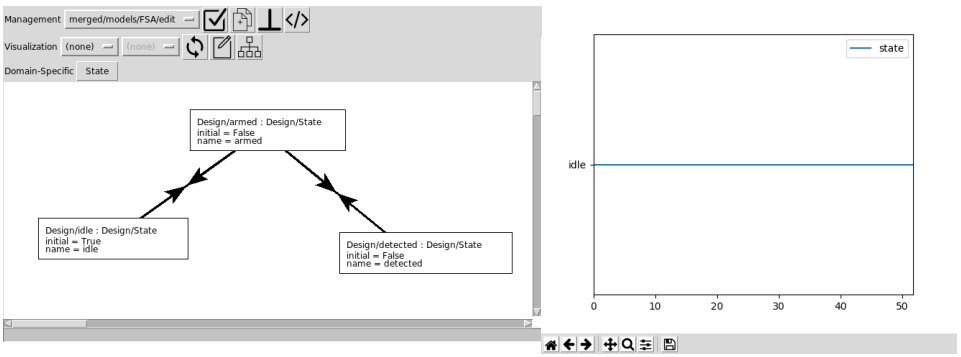


Figure 6.25: Live modelling for FSAs, before change.

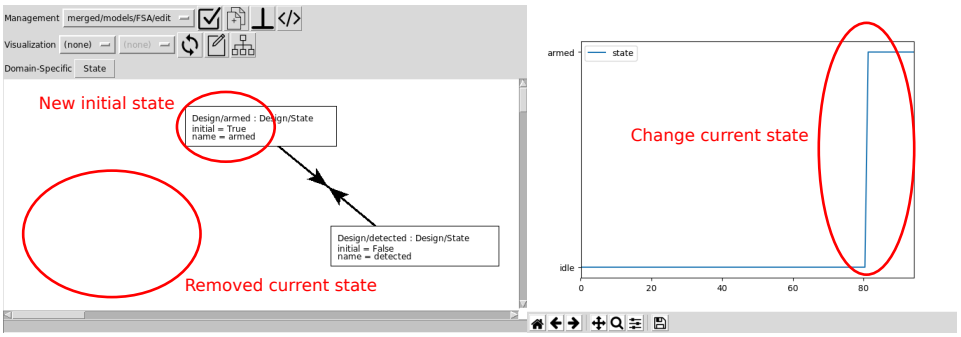


Figure 6.26: Live modelling for FSAs, after removing current state and setting new initial state.

Continuous Time Causal Block Diagrams

The implementation of our CTCBD live modelling environment is shown in Figure 6.29. It is identical to the DTCBD live modelling environment, but now we have access to the derivator and integrator blocks. To the user, it is indistinguishable whether this live modelling functionality was provided by using an operational or denotational semantics approach. Similarly, the simulation viewer from DTCBDs is reused.

Up to time 60, the simulation executes the model shown in Figure 6.29, showing the results on the trace in Figure 6.30. We notice the harmonic oscillator behaviour that is expected of such a system. At time 60, however, the CTCBD model is altered by changing the value of constant g from 10 to 30, effectively being a sudden increase in gravitational force. This has an immediate effect on the simulation trace, as shown in Figure 6.30 after time 60: instead of having a decreasing velocity, the velocity starts increasing again. Results stay continuous, though a difference in behaviour is clearly observed at the point in simulation where the change was made.

Evaluation

Given that our generic tool could be used for three different domain-specific languages, while all using the same (parameterized) FTG+PM model, we believe our approach to

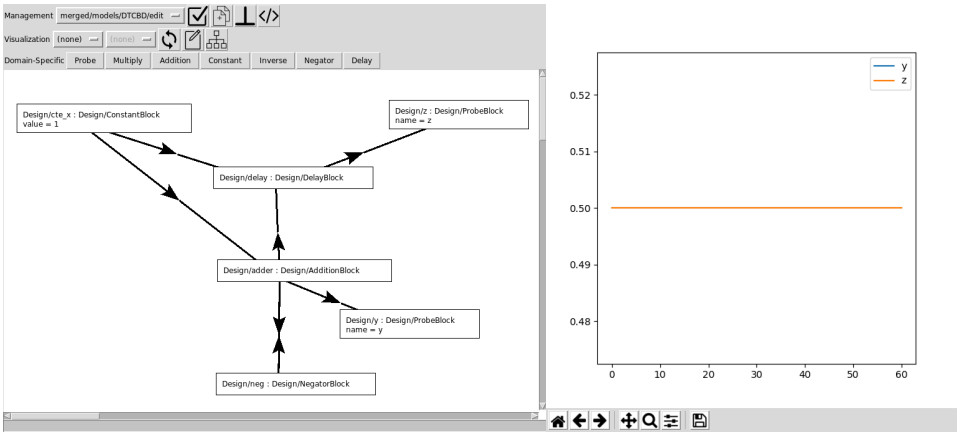


Figure 6.27: Live modelling for DTCBDs, before change.

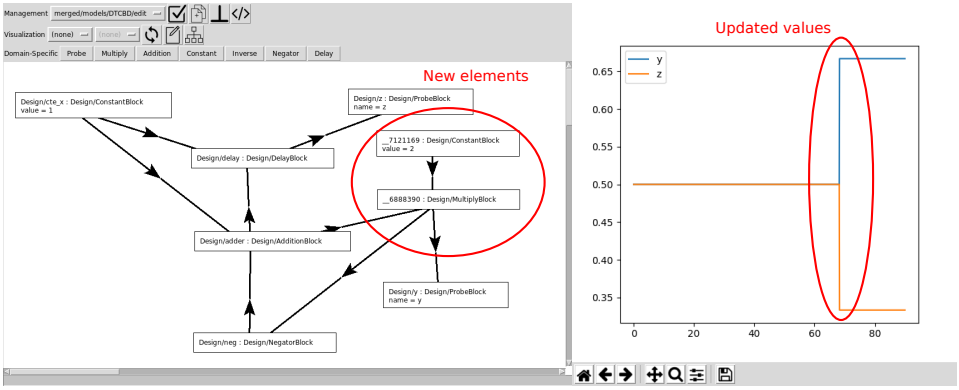


Figure 6.28: Live modelling for DTCBDs, after adding and connecting the multiplication block.

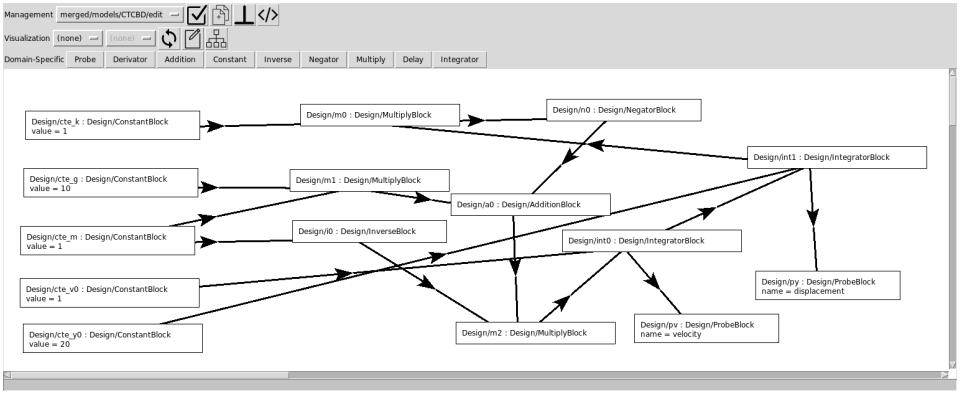


Figure 6.29: Implementation of live modelling for CTCBDs.

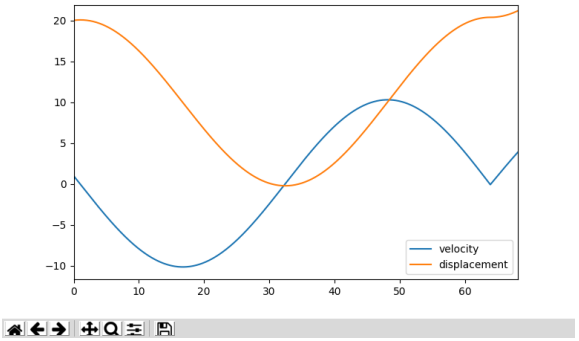


Figure 6.30: Simulation trace, where the constant "g" is changed at around time 63.

be applicable to a wide variety of modelling languages. Indeed, our structured approach required no modification for these three types of semantics, making us believe that it can be applied for other formalisms as well. We can therefore assume that our approach provides structure to the currently ad-hoc process of making a language live.

Sanitization was the only activity that was further required, but its logic was previously described. The goal of the sanitization function is conceptually clear: combine the currently executing model (with state information) with an uninitialized runtime model. In the limit, this sanitization function can be seen as an advanced initialization function, which can take an existing simulation model as input. We can therefore assume that our approach can make existing languages live with little additional work for the language engineer.

6.2.6 Related Work

Our contribution provides a language-neutral overview of liveness, thus enabling liveness for (domain-specific) executable modelling languages. Two categories of related work exist: live programming and (live) executable modelling.

In the live programming domain, the concept of liveness is well-studied. One of the most important distinctions between different approaches is how they handle time: a distinction is made between *real-time* and *recorded event* [194]. In real-time mode, the past is left unaltered, and only future executions of the code are influenced. This is often termed *fix and continue*, as implemented by Lisp [249], Smalltalk [123], Erlang [18], and SELF [283]. In recorded event, all past input events are recorded and replayed, resulting in a completely new history. This is implemented in languages such as ElmScript [74] and YinYang [194]. We only implement the real-time live modelling approach, as recorded live modelling has been shown not to be ideal for simulation [194]. Nonetheless, further investigation into recorded event live modelling might be interesting for other types of languages.

A lot of work is spent towards making live programming usable. This requires research as to which representation is most usable, such as textual or graphical languages [102, 128, 193, 219]. Therefore several kinds of languages have been made live: graphical languages such as VIVA [276] and Flogo [132], textual languages such as ElmScript [74] and Smalltalk [123], and hybrid languages such as Subtext [102]. Our approach does

not commit itself to textual, graphical, or hybrid languages. It is implemented on the abstract syntax of models, and does not require a specific visualization. If required, our live simulator can be coupled to multiple interfaces, possibly with different representations (e.g., textual, graphical).

Another important usability aspect of live programming is the need for immediate feedback to the user [276], resulting in the need for effective visualization and tight latency constraints [193, 263]. Latency is considered harmful when it becomes too large, with the threshold being defined somewhere between $50ms$ [194] and $500ms$ [193]. For this reason, a lot of work has focused on optimizing specific aspects, such as incremental compilation [194] and code hotswapping [107]. Our framework focuses exclusively on the functional requirements of live modelling, without considering performance, visualization, and so on. While these concerns are certainly important, we consider them as future work.

Many challenges related to live modelling are tackled only for specific cases or specific languages. An example issue is the question how the state needs to be retained [107, 268], and what needs to be recomputed [57]. Making an existing programming language live is often done through ad-hoc modifications, often turning liveness into a black art [55]. With our approach, we provide an overview of the steps required to make an executable modelling language live. And while not fully automated, since some domain information remains necessary, the approach becomes structured and easier for language engineers to understand and implement.

In the modelling domain, the focus has primarily been on the theoretical foundations of (meta-)modelling [169] and how (domain-specific) modelling can help developers [154]. Nowadays, focus starts shifting to model execution [195] and debugging [189]. And whereas model debugging is often formalism-specific, such as for Causal Block Diagrams [320] and Parallel DEVS [300], recently new approaches have been developed that try to (partially) automate the addition of debugging to formalisms [293]. Advanced tracing facilities for domain-specific languages have been developed [51], which enable omniscient, or backwards-in-time debugging [50]. Closer to our approach is [289], in which the author explores how executable modelling languages can be made live with "semantic deltas". The system is capable of translating source program modifications (so-called deltas) to operations on the running code. While they present a prototype demonstrating the approach, it does not present a structured way to add live modelling to modelling languages. Similarly, another approach is based on textual differences [302], where existing textual difference algorithms are leveraged to update the executing model. While that approach is also relatively generic, it focuses exclusively on textual languages, and is only evaluated in the context of one kind of finite state automaton. Since live modelling is rarely implemented, or at best in an ad-hoc way, we contribute by providing a general framework for merging liveness into existing modelling languages, paired with an implementation for two example languages.

Similar to reflection and code hotswapping, the modelling community is starting to acknowledge the existence of models at runtime. These models, however, are mostly used for self-managing systems [199, 233], and do not directly apply to live modelling. Specifically, models at runtime make the changes internally, as a part of pre-defined, correct behaviour. Live modelling, on the other hand, makes changes due to external operations, knowing that some part of the model may be incorrect. Additionally, models at runtime techniques are

used to express dynamicly changing systems, whereas live modelling is used for modifiable systems (e.g., for debugging or education). Due to this mismatch in application domain, their requirements severely differ. For example, models at runtime do not need to cope with changes at the design model, but applies changes on the full runtime model, rendering sanitization unnecessary.

Finally, model evolution [108], and in particular language evolution, has similar challenges to code hotswapping. When swapping code, but retaining the state, the old state might not be understandable for the new code operating on it [107]. Similarly, language evolution tries to tackle the problem of existing models not being updated after a language change. Sanitization, as part of model co-evolution [197], tackles such changes semi-automatically. Our sanitization approach is similar, as we also need to adapt a model under execution to an evolving model.

Summary

We have argued in favour of live modelling: live programming transposed to modelling. While some approaches exist that already support live modelling, they do so in an ad-hoc way that cannot be transposed to different (domain-specific) languages. We deconstructed the live programming process and reconstructed this on top of modelling, pushing all live modelling logic into a new *sanitization* activity. Using our approach, adding liveness to (domain-specific) modelling languages becomes more structured and reproducible, though still necessarily manual. As an example of our approach, we have applied this framework to three languages: Finite State Automata (operational semantics with breaking changes), Discrete Time Causal Block Diagrams (operational semanticsw ith non-breaking changes), and Continuous Time Causal Block Diagrams (denotational semantics). All these modelling languages have distinct characteristics, demonstrating that our approach is widely applicable. For each, a new sanitize activity was defined, while reusing all other operations and processes, which was sufficient to support live modelling.

6.3 Concrete Syntax

The language engineer is the second type of user we consider. Recall that one of the responsibilities of the language engineers was to create intuitive languages, correctly representing the problem domain. In particular, this means that the languages must be as close to the problem domain as possible, in both abstract and concrete syntax. Many limitations exist on the concrete syntax of languages, making languages less intuitive than they should be. The core problem is that many tools, if they support concrete syntax at all, restrict themselves to an icon-based visualization. While this is helpful in many cases, icons alone are often not sufficient.

To allow the language engineer to build usable languages, we introduce a multi-paradigm modelling approach to concrete syntax. Using this framework, many of the existing limitations are lifted and language engineers gain more power when defining concrete syntax. The usability of the created concrete syntax itself is not considered: it is still up to the language engineer to consider the options available and use them wisely.

6.3.1 Motivation

Domain-Specific Modelling Languages (DSLs) are defined by their abstract and concrete syntax [161, 299]. The abstract syntax defines the concepts of the language, which can be instantiated and used as the building blocks of models. For example, the abstract syntax of UML Class Diagrams defines concepts such as *Class*, *Association*, and *Attributes*. The concrete syntax defines the visualization, or rendering, of these abstract syntax concepts. For example, the concrete syntax of UML Class Diagrams defines the mapping of a Class instance to a rectangle with the name of the class on top and a list of all attributes below it. Significant restrictions exist in current tools for the definition of concrete syntax, thereby restricting the language engineer, responsible for creating intuitive languages.

Code-based solutions (i.e., tool plugins) are now often used to implement advanced concrete syntax functionality. While feasible, the creation of plugins is not always for the faint-hearted [262], as it relies on tool details (e.g., API) and advanced functionality is non-intuitive to express. Additionally, creating the concrete syntax is part of the job of the language engineer, who is not necessarily an expert in tool plugin creation. To address these problems, we present a different angle of attack, where we apply the Multi-Paradigm Modelling (MPM) approach to concrete syntax. We identify several limitations in the concrete syntax of state-of-the-art approaches, which are now addressed with (coded, language-specific) plugins. All these limitations become easy to solve using our MPM-based approach, without the need for plugins.

We identify five common limitations: (1) A **single front-end** (or visualization tool) is provided, which is largely aware of the concepts of (meta-)modelling. Existing visualization libraries therefore require a lot of additional code, as these (meta-)modelling concepts need to be introduced. (2) A **single representation** is used for all languages, such as one consisting of rectangles and lines, often arranged in a graph-like manner. While these can be used as primitives for many types of visualization, some models are ideally expressed using a plot, or completely different modes of perceptualization. Note the use of the term *perceptualization*, as we do not wish to limit ourselves to visual representations of models, but want to include, for example, text and sound as well. (3) A **single mapping** to the representation is used, such as to UML Object Diagrams, which can be used for all (graph-based) models, but is seldom the most appropriate. Even when a domain-specific concrete syntax is defined, it is often restricted to only one such mapping. (4) **No extra concrete syntax operations** are available, such as domain-specific lay-outing [101], which aids users in understanding the model. As these algorithms are domain-specific, they must be part of the specification of the domain-specific language. (5) A **one-to-many** mapping between abstract syntax and the visualized model is used, as an icon definition is used. While this often suffices, many-to-many mappings offer additional possibilities to the language engineer.

To show the need for more flexibility in concrete syntax definitions, we use the Causal Block Diagrams (CBD) language [61] as a running example. While this language can be created and used in current tools, its concrete syntax can not easily be implemented as we would like. For an optimal interaction between the modeller and the model, several extensions to concrete syntax are proposed next. The previously mentioned restrictions are now elaborated on in the context of this motivating example: the CBD language. Note that our contribution lies in the explicitly modelled framework for concrete syntax, and not in the extensions offered for this specific domain specific language.

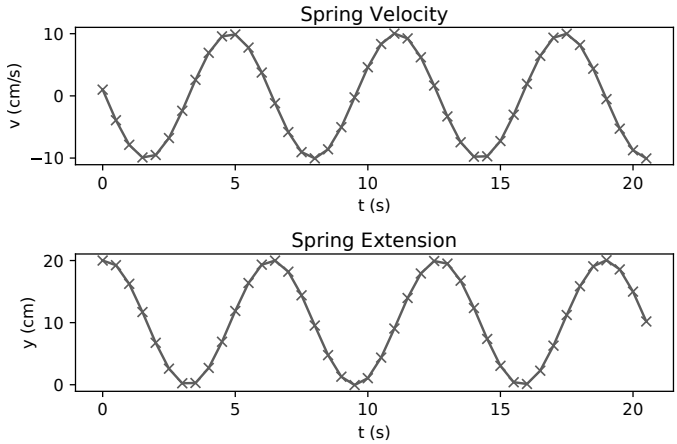


Figure 6.31: Plotted trace of the CBD model, with $k = 1$, $m = 1kg$, $y_0 = 20cm$, $v_0 = 1\frac{cm}{s}$, and $g = 10\frac{cm}{s^2}$.

Our example instance is the same as in the previous section, more specifically that of a mass suspended by a vertical spring, as shown in Figure 6.14a.

The semantic domain of CBDs is a trace language whose instances contain the values in the probe blocks, paired with the simulation time at which the value was recorded. An appropriate perceptualization of a real value changing over time is a plot, as shown in Figure 6.31. From this figure, the evolution of the value throughout time becomes immediately obvious. In our case, there are two plots: for the velocity of the mass (v) and the current position (y).

As previously introduced, many limitations currently exist in the perceptualization and rendering of models. For each of these limitations, we present a potential requirement of the CBD concrete syntax, and how current tools fail to adequately address them. In our related work subsection, we further discuss specific tools and the techniques they use.

Multiple GUIs When modelling CBDs, different users might have different preferences in how they interact with their model. Some users prefer an online browser-based application, requiring no installation nor local code execution, while other users prefer an offline application which executes locally. Nonetheless, the model should be visualized in the same way. As these users want to collaborate, they should share the same back-end, while their front-ends are different.

Although many (meta-)modelling tools explicitly make the distinction between a back-end and front-end, or expose a modelling API, the distinction between front-end and back-end is often not as expected. Most of the time, the front-ends still need to be aware of most meta-modelling concepts, as they receive the abstract syntax model, the metamodel, and a concrete syntax definition. Changes to the model are then performed in the front-end, and only the abstract syntax changes are propagated to the back-end. Multiple front-ends therefore duplicate this modelling code, while it should only be concerned with the binding to the platform (e.g., TkInter).

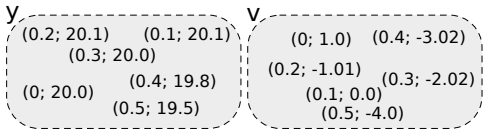


Figure 6.32: Graphical representation of the trace in Figure 6.31.

Multiple Perceptualization Formats Visualizing CBDs is completely different from visualizing their semantics. The semantics of a CBD, expressed as a trace of its signals, is ideally shown as a plot, instead of a graph-like structure. A possible rendering of a trace with the same perceptualization format as the CBD model is shown in Figure 6.32. Clearly, the trace is better visualized as a plot, previously shown in Figure 6.31: the plot immediately shows the oscillating behaviour, which cannot easily be derived from the set of tuples.

While different front-ends exist today, most are restricted to a graph-like or text-only representation of the models. Other perceptualizations might reason about different concepts, such as datapoints (for plots) or music notes (for sonification), instead of graphical primitives.

Multiple Mappings The ideal visualization of a CBD model depends on the domain expert looking at it, even when the visualization is relatively similar (e.g., both block-based). Some elements might have a different icon attached to them, depending on the background of the user. For example, users with a Simulink® background are familiar with the symbols $1/s$ for an integrator, and Σ for an addition block. Other users might prefer the symbols $\int$ and $+$, respectively. A visualization with an alternative set of icons is shown in Figure 6.33.

Even though many tools nowadays support the definition of custom icons for a language, there is often only one possible visualization attached to it. As such, when a different visualization is required, the complete model, including abstract syntax, must be copied. While some tools allow for workarounds, such as defining both icons and only showing one, depending on a configuration option, this is not an elegant solution.

Lay-Outing The ideal lay-out of CBD elements is closely related to its dataflow. If the flow goes left-to-right, with the exception of feedback loops (e.g., Figure 6.33), the semantics is easier to interpret than if the position seems random, as in a circle lay-out [261] (e.g., Figure 6.34).. The flow of the data, and therefore the ideal lay-out, is specific to CBDs, as it depends on the topological sort of the dependency tree [61]. This is specific to CBDs and should therefore not be hard-coded in either the back-end or front-end: it should be defined and maintained by the language engineer.

While current tools often implement generic lay-out algorithms, such as circle and spring lay-out, they have no support for lay-out algorithms provided by the language itself (i.e., domain-specific lay-out algorithms). Lay-outing can be generalized as a "post-processing operation" on the rendered model, where the visualized model is reordered. There is thus a need to define algorithms on the rendered model, ideally included in the concrete syntax model.

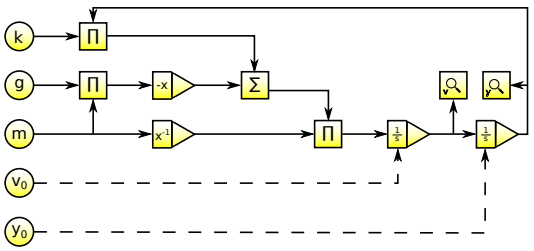


Figure 6.33: Using different set of icons.

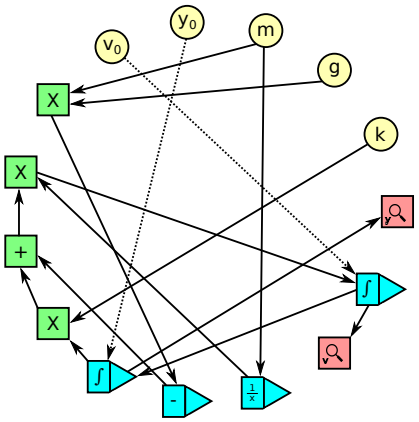


Figure 6.34: Circle lay-out of the same model.

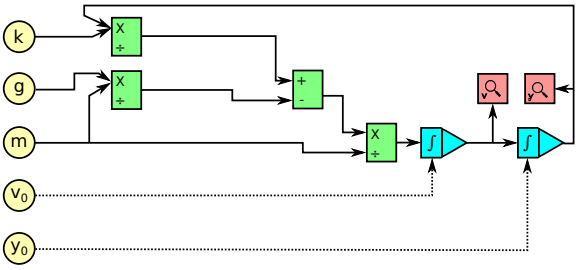


Figure 6.35: Alternative representation using a more complex mapping.

Many-to-many Mapping and Parsing While we have previously allowed for multiple mappings, thereby allowing for a single element to be visualized in multiple ways, the modeller might have additional preferences. For example, CBDs are sometimes visualized with a conjoined addition/subtraction block (e.g., in Ptolemy/Kepler [13]): a single block has an addition and subtraction port, where all signals are summed, but the signals on the subtraction port are negated first. This is syntactic sugar for a single addition block, with negation blocks for each input on the subtraction port, as shown in Figure 6.35. Whichever representation is used depends on the domain expert, though we want the abstract syntax to be identical, independent of the used representation.

While this problem seems highly related to the multiple mappings problem, it is fundamentally different: the conjoined addition/subtraction block is a single concrete syntax element with multiple abstract syntax elements underlying it. Indeed, each connection to the subtraction port has a (hidden) negator block in the abstract syntax. While it is possible to change the abstract syntax, this would create problems for the other operations, where the negation block is explicitly present. The problem is therefore the restriction of many tools to a one-to-many mapping: a single abstract syntax element is rendered by several concrete syntax elements, independently of other abstract syntax elements. A possible workaround is the introduction of an intermediate language, which expands or collapses the addition/subtraction block, though such an intermediate language causes additional

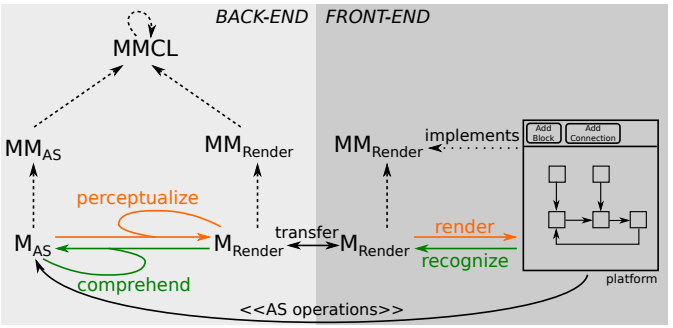


Figure 6.36: Overview of the approach.

consistency problems.

6.3.2 Approach

We now present our multi-paradigm modelling approach to concrete syntax, where we explicitly model all aspects.

Our approach makes a clear distinction between the responsibilities of the back-end and front-end. The back-end is responsible for all (meta-)modelling related concepts, including how models are perceptualized and comprehended. The front-end is responsible only for how this perceptible model is rendered using a specific platform, such as TkInter. Instead of transferring the abstract syntax of the model (using domain-specific concepts, such as *Constant*), the back-end transforms this model to the MM_{Render} language (which uses perceptualization concepts, such as *Ellipse*).

Our approach is centered around four activities, as shown in Figure 6.36: *Perceptualization*, *Rendering*, *Recognition*, and *Comprehension*. Our approach is independent of how these activities are implemented (e.g., in code, using model transformations, or manually).

We now elaborate on each step of our approach, where we link to a minimal example in the context of CBDs, shown in Figure 6.37. We start at the top left in the figure, with M_{AS}. M_{AS} is first (1) perceptualized, resulting in an M_{Render}. For example, instances of the *Constant* class are translated to a *Group* containing an *Ellipse* and *Text* instance. This M_{Render} is (2) transferred to the front-end in some way, which is independent of our approach. In the example, a JSON serialization of the source model is shown. The front-end has a copy of M_{Render}, which is (3) rendered for that specific platform. For example, all instances of *Ellipse* are iterated over, and a `create_oval` TkInter function is invoked. The TkInter front-end listens to user events (e.g., mouse clicks), thereby (4) altering the rendered model. For example, the text entry "1" is altered to "2". Such changes are (5) recognized (e.g., via callbacks), resulting to changes on M_{Render}. For example, the *Text* instance has its attribute *text* updated to "2". Changes are (2) transferred to the back-end again, this can be incremental or overwrite the complete model. Finally, the new M_{Render} is (6) comprehended, thereby changing M_{AS}. For example, the changed text results in an update to the value of the constant block. Each of these steps is further elaborated on next.

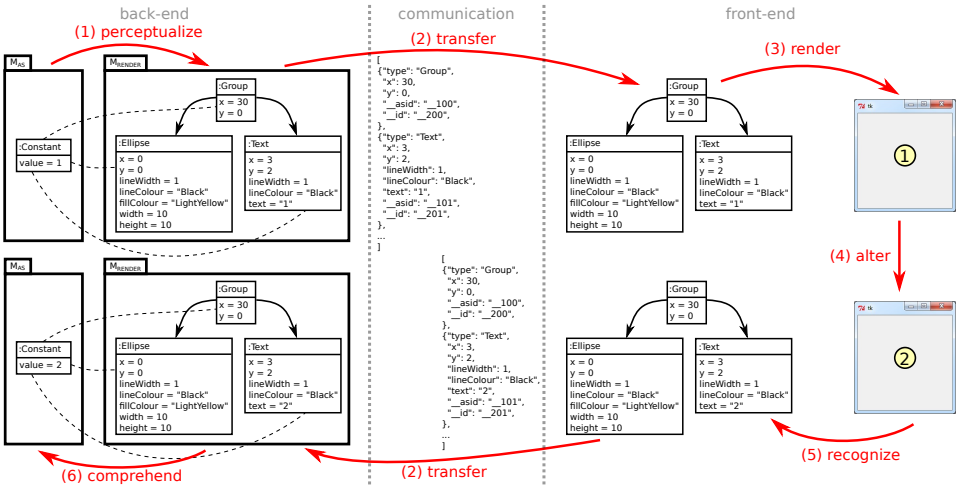


Figure 6.37: Overview of the approach with an example for CBDs.

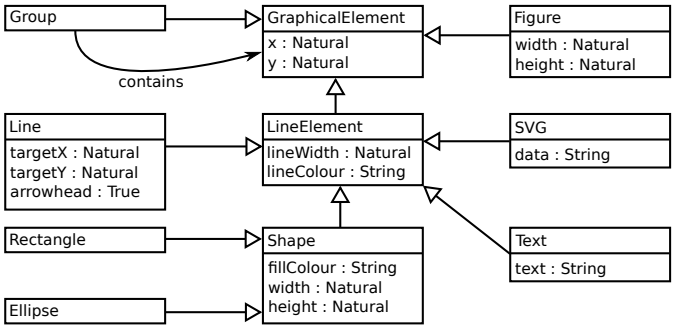


Figure 6.38: MM_{render} for graphical visualization.

Perceptualization

The first step to our approach is perceptualization, where a model in a domain-specific language MM_{AS} is mapped to a perceptualization language MM_{Render}. This defines how the model is presented to the user. For each language that we want to visualize, it is important to define a perceptualization activity, which is the concrete syntax definition.

This activity needs to map to an MM_{Render}, which defines the mode of presentation to the user. MM_{Render} defines the platform primitives that can be used, such as *Ellipse*, *Rectangle*, and *Line*. In our example we focus on graphical languages, as this is easiest to present on paper, and therefore our MM_{Render} is defined as in Figure 6.38. Note that this MM_{Render} is not yet linked to any specific platform, such as TkInter or Scalable Vector Graphics (SVG). The used concepts are generic to many graphical visualization libraries.

Our approach is not restricted to any specific MM_{Render}, although we demonstrate our approach using a metamodel for graphical visualization. It is straightforward to come up with different MM_{Render} specifications, such as one for plots (e.g., for signal traces), text (e.g., for action language), or even sound (e.g., for music sheets [265]). We envision a small

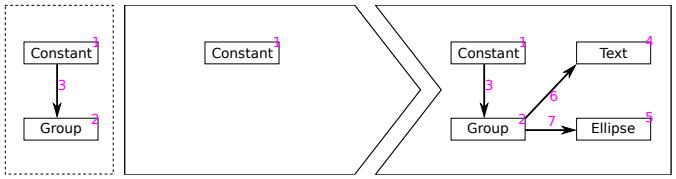


Figure 6.39: Example rule for CBD perceptualization.

library of different kinds of MM_{Render} to capture all necessary perceptualizations. Of course, a front-end should also be defined which can render models in that language.

Traceability can be constructed between M_{AS} and M_{Render}, to be used for incremental perceptualization, where we only perceptualize elements in M_{AS} that have no associated elements in M_{Render} yet. This is the reason for the loop in Figure 6.36, where perceptualization takes in both M_{AS} and the current M_{Render}. It remains up to the language engineer whether or not to use incremental perceptualization.

In our example, we transform the single CBD instance of *Constant* to instances of *Group*, *Ellipse*, and *Text*, conforming to the MM_{Render} metamodel in Figure 6.38. This defines how constant blocks are to be presented to the user: as a group of an ellipse and some text. We defined this activity using model transformations. An example model transformation rule is shown in Figure 6.39, which creates a *Group*, *Ellipse*, and *Text* instance for each *Constant* element that not yet has an associated group. The values of their attributes are hidden due to space restrictions, but are mostly trivial (e.g., the colour of the ellipse and value of the text). In a model transformation rule, the Left-Hand Side (LHS) pattern is matched in the model, and is replaced by the Right-Hand Side (RHS), unless the Negative Application Condition (NAC, shown in the dashed rectangle) also matches. The (purple) numerical annotations link elements in the LHS to elements in the RHS.

Model Transfer

As there is an explicit difference between the back-end and the front-end, there needs to be a way to transfer the models. We want this to be as general as possible, as both the back-end and front-end could be physically distributed and implemented in different programming languages. In our example, the model is serialized using JSON, and transferred over network sockets. Nonetheless, our approach is independent of the implementation details of model transfer, and we therefore do not elaborate on this aspect. It is only important that an exact copy of M_{Render} is present on both the back-end and front-end; this can be achieved in many different ways.

Note that, thanks to our approach, only models in the MM_{Render} language must be transferred, potentially allowing for additional optimizations in the serialization.

Rendering

When the M_{Render} arrives at the front-end, it needs to be presented to the user. This is done by mapping the concepts of M_{Render} to the platform operations responsible for the presentation. As such, the front-end's interface is described in a platform-independent way using MM_{Render}. It is thus important that the front-end and back-end agree on the

same MM_{Render}. Rendering can be seen as a transformation from concepts in MM_{Render} to concepts in the platform.

While our approach explicitly represents both M_{Render} and MM_{Render} in the front-end, this does not necessarily have to be the case. For example, the front-end could just iterate over the JSON serialization it gets in, directly invoking platform functions. And even while the models are not explicitly present in the front-end, the front-end still makes implicit use of these models and the back-end ensures well-formedness.

In our example, the front-end maps concepts such as *Ellipse* to the `create_oval` TkInter function, also translating the attributes to arguments for that function. The complexity of the mapping on how close the concepts of MM_{Render} match those of the platform. For example, if a platform does not support rectangles, elements of the *Rectangle* class have to be mapped internally to four seperate lines (or whatever operation the platform provides).

Altering

Some front-ends allow altering the rendered model in some way. Straightforward examples are moving around elements, changing their size, and so on. Such changes occur in the platform, and are based on platform events (e.g., button press, mouse move, mouse click), which need to be mapped to model operations. As the detection of such events is highly platform-dependent, and can be considered an implementation detail, we do not elaborate on this. For our approach, it is only important that the rendered model can be altered, as we are independent of how these changes actually occur.

Even though simple operations are common, altering the model can happen in any way, for example through sketch interpretation [77, 209], where sketches are recognized as changes in the platform (e.g., a drawing of a circle is mapped to the TkInter circle concept).

Recognition

When changes are made to the rendered model, these changes have to be propagated to the M_{Render}, as this is the common exchange format between back-end and front-end. While this mapping is often trivial, it depends on the match between MM_{Render} and the platform concepts. For example, for a trivial mapping, moving a rectangle in the platform merely maps to moving that same rectangle element in M_{Render}. For a complex mapping, however, the rectangle might be a set of lines in the platform, where moving one of these lines affects the three other lines as well.

Recognition does not attach semantics to the change. Indeed, changing the value of the text merely alters the text value, and the associated constant block still has the value 1. As such, recognition is limited to syntactical changes.

In our example, the mapping is trivial: updating the text value in the platform merely requires us to update the *text* attribute of the *Text* instance in M_{Render}.

Comprehension

Comprehension maps changes on M_{Render} back to changes on M_{AS}. As such, it attaches semantics to the change that was made. Note that this operation often makes use of the traceability information that was previously created during perceptualization, as it needs

to map between both formalisms. Therefore, comprehension can make use of the original M_{AS}, being the reason for the loop in the overview figure.

Often, a front-end only allow syntactical changes that have no influence on semantics. For example, moving an element of a topological formalism changes the *x* and *y* attributes in M_{Render}, though it has no effect on the semantics of the model. In many cases, therefore, comprehension is skipped completely. Nonetheless, it is an essential activity in the context of free-hand editors, where all changes are made purely in concrete syntax.

The distinction between recognition and comprehension is important. For example, recognition recognizes when a rectangle is dragged to a different location (changing its *x* and *y* attributes), and comprehension comprehends that this implies containment (creating a *Containment* link). In contrast to perceptualization, comprehension might fail if the user creates a structure that cannot be comprehended (i.e., a *parsing error*). While we are sure that the modified M_{Render} conforms to MM_{Render}, it does not necessarily represent a comprehensible model (e.g., a circle has no meaning in CBDs without a text value in it).

In our example, comprehension maps the text value of the *Text* element back to the value of the *Constant* block. Note that this is one of the only changes on concrete syntax that would have any semantical effect. For example, altering the *x* and *y* attributes of any of these elements would have no semantical effect, as CBDs are a topological formalism. When the *Text* element is deleted altogether, comprehension fails.

6.3.3 Evaluation

We now evaluate our approach by considering the various dimensions of flexibility achieved. Our research questions are directly related to these dimensions.

Dimensions of Flexibility

With our approach explained, we present how this approach addresses the various restrictions of existing tools. For each restriction, we explain how our approach is flexible enough to support it, applied to our motivating example.

As we did not code our approach, many of these dimensions of flexibility are just the creation of a new model, in which meta-modelling tools are specialized. The presented dimensions of flexibility can therefore be explained at a high level of abstraction, without going into implementation details. This would not be the case for a plugin-based approach, for example, as we would have to rely on tool-specific API calls.

Multiple GUIs The first restriction was related to having multiple front-ends, possibly implemented in different implementation languages, though all with similar semantics. We addressed this problem by presenting the MM_{Render} model as the "interface" for model rendering: all front-ends must accept the same set of models. As long as the back-end and front-end agree on a certain MM_{Render}, specified in the back-end, all front-ends that implement it are supported. In contrast to other tools, where the front-end is offered some kind of fixed modelling API on abstract syntax, our front-end only receives a serialized model, in a known format, which it must render as-is: all processing has already been done.

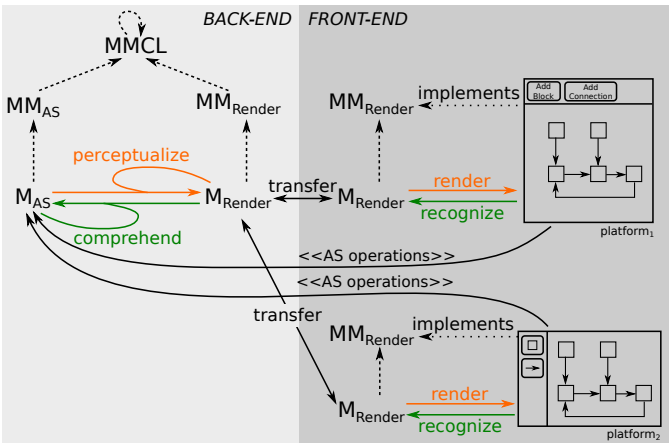


Figure 6.40: Approach with multiple GUI front-ends.

The back-end is completely independent of the front-end and, subsequently, the platform used for rendering. This is shown in Figure 6.40, where the same M_{Render} and MM_{Render} is used for two different front-ends, rendering the same representation of the model.

For CBDs, we implemented a front-end in Python/TkInter and JavaScript/SVG. Both are similar in use and visualization, though their underlying mapping to the platform drastically differs. There is still much freedom left in the front-end, specifically for elements not defined in MM_{Render}, such as the supported operations (e.g., zooming, scaling) and interaction with the user (e.g., mouse-driven, keyboard-driven).

Multiple Perceptualization Formats Since our approach explicitly models MM_{Render}, it is possible to have several variants, each defining another format. Each front-end merely ensures that its MM_{Render} is comprehended by the back-end, and can from then on visualize models in that language. A different MM_{Render} often requires a different front-end, though this is not required. For example, a TkInter front-end can visualize a text-only MM_{Render} as a TkInter text widget.

Figure 6.41 shows this in the context of our CBD example, where we have two rendering formats: one for graphical models (MM^{Graph}_{Render}), and one for plots (MM^{Plot}_{Render}). Each MM_{Render} has its own front-end. Both front-ends are connected to the same back-end and share the same models and API to these models. Through this API, a graphical front-end receives a model conforming to MM^{Graph}_{Render}, and the plotting front-end receives a model conforming to MM^{Plot}_{Render}.

Multiple Perceptualizations Since the mapping from MM_{AS} to MM_{Render} is explicitly modelled, it is possible to change it, or have multiple. Any mapping is fine, as long as it generates a valid instance of MM_{Render}, and can therefore be rendered. These mappings can target different versions of MM_{Render}, as was already shown in the previous point, but can also go to the same MM_{Render}.

Figure 6.42 shows this in the context of our CBD example, where we have two mappings

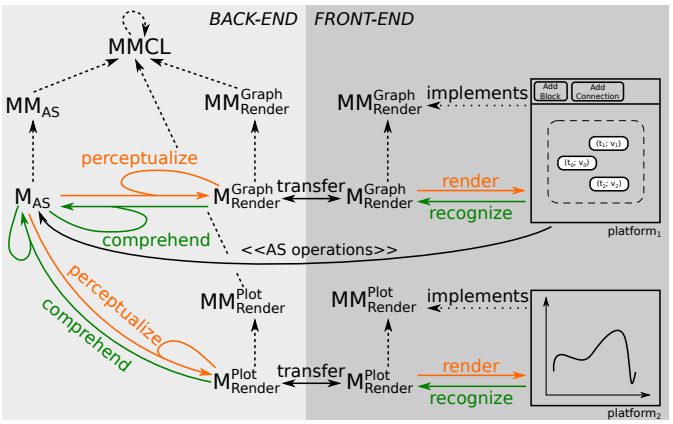


Figure 6.41: Approach with multiple MM_{Render} models.

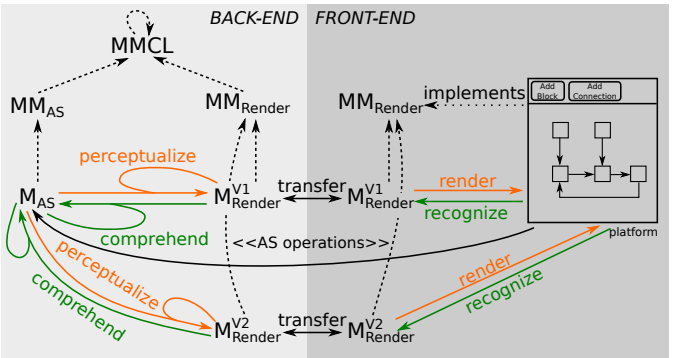


Figure 6.42: Approach with multiple mappers to the same MM_{Render}. The same tool is used for both models, though different instances.

to the same MM_{Render}. One defines the integration block icon as a rectangle with $1/s$ in it (MM^{V1}_{Render}), whereas the other defines it using a triangle and the $\int$ symbol in it (MM^{V2}_{Render}). Both mappings are equally correct and can be used interchangeably: all changes on one representation are automatically mimicked on the other representations, as they share the same M_{AS}.

Lay-outing Lay-outing is an additional operation executed after perceptualization, as we need to operate on the current visualization. Therefore, it is often shifted to the front-end completely. In our approach, the perceptualized model is available in the back-end, where the lay-outing can happen using, for example, model transformations. This not only makes it possible to share the same lay-out algorithms between front-ends, but also allows domain-specific lay-outing algorithms. For practical reasons, the lay-out algorithm, and any other pre- or post-processing operations, are implemented as part of the perceptualization phase.

For our CBD example, we can implement a new domain-specific lay-out algorithm as part

of the perceptualization. When new elements are added, users can add them wherever they want, but they will automatically be placed at the ideal location in the CBD model. With lay-outing happening at the back-end, all users sharing the same perceptualized model will also see the lay-out propagated.

Many-to-Many Perceptualization As our mapping for the perceptualization and comprehension is any kind of operation, we can use any executable language to define it in. In contrast to icon definitions, we can map multiple abstract syntax elements to multiple concrete syntax elements, as the mapping itself is generic. This can be used during perceptualization to create complex rules that cannot be expressed with the usual icon definitions: multiple abstract syntax elements are condensed into a single icon. Thanks to the use of traceability links in our approach, from M_{Render} to M_{AS}, it is also possible to incrementally update the concrete syntax, by linking previously rendered elements.

For our CBD example, we are able to utilize model transformations to map elements from the source language (M_{AS}) to elements in the target language(M_{Render}). In general, model transformation language are not limited to a one-to-many mapping, in contrast to most icon definition languages.

Research Questions

We distill our motivating example into five research questions:

- **R1**: Can new front-ends be implemented fast?
- **R2**: Can models be perceptualized in different ways?
- **R3**: Can multiple perceptualizations be defined?
- **R4**: Can domain-specific lay-outing be defined?
- **R5**: Can many-to-many perceptualizations be defined?

R1: Lightweight Front-ends We have implemented two separate front-ends, for two different platforms: TkInter and Matplotlib, both using Python. The Matplotlib front-end only visualizes the model and does not offer any manipulation operations. The TkInter front-end includes basic concrete syntax operations, such as moving around elements, and basic abstract syntax operations, such as modifying attributes. Each front-end was implemented by a different developer, familiar with the platform. Each individual front-end took less than one day to implement up to the point where they could visualize the models, exactly as received from the back-end. Each front-end has a small code base: approximately 250 lines of Python code for the front-end with Matplotlib, and 350 lines for the front-end with TkInter. For both front-ends, no (meta-)modelling information had to be coded, except for the implementation of MM_{Render}. This can be considered fast for front-end development, which usually takes a significant amount of time. In our case, perceptualization was only defined once in the back-end, instead of once for each front-end.

R2: Different Perceptualizations Using the two previously implemented front-ends, we have also shown the feasibility of different perceptualizations. The first front-end provides a plot-based perceptualization of a trace model. In this perceptualization, the model is visualized as a graph, and all operations, such as zooming, are provided natively by the Matplotlib platform. The second front-end provides a graphical perceptualization of the original CBD and resulting trace model. In this perceptualization, we rely on the TkInter visualization primitives. The trace model can therefore be perceptualized in two significantly different ways.

R3: Multiple Similar Perceptualizations Using the previously implemented graphical front-end, with TkInter, we have implemented two different perceptualizations as model transformations. This front-end therefore has a drop-down menu for the model to show, and a drop-down menu for the available perceptualizations. Both are automatically populated by querying the back-end. The same model can therefore be visualized with two slightly different transformations.

R4: Domain-Specific Layouting We have implemented a simple lay-out algorithm in the perceptualization transformation. Combined with the two different perceptualization transformations, we were able not only to alter the icons of the different concrete syntax elements, but also to change their relative position. As such, when switching from one perceptualization to the other, the model not only changes its icons, but the position of these icons changes as well.

R5: Many-to-many Perceptualization As our approach is based on generic activities, it stands to reason that we can support many-to-many perceptualization. A simple many-to-many perceptualization was implemented, as presented before in the motivating example. After the usual icon mapping, mapping an addition block to a rectangle with the addition symbol in it, additional model transformation rules are added to search for a negation block that is connected to the addition block. When such a pattern is found, the concrete syntax representation of the negation block is removed, and the connection is redrawn to the negated input port of the addition block. As such, a one-to-many mapping between M_{AS} and M_{Render} is shown to be possible.

Threats to Validity

For **construct validity**, our primary threat is the measures used for R1. We used two measures: the time needed to develop the tool, and the number of lines of code. Development time highly depends on the skill of the developer and the familiarity with the used libraries. To minimize the time needed to get familiar with the libraries, developers were familiar with the library they had to use up to the level that they had no technical problems. The number of lines of code is not too reliable to determine the difficulty of writing the front-end. The codebase of the two front-ends mostly consists of linear code and does not include non-trivial algorithms. For example, out of the 250 lines of code for the plotting front-end, 50 lines are dedicated to the translation of terminology (e.g., “solid” line types in MM_{Render} to “-” in Matplotlib).

For **external validity**, our primary threat is the application to only a single language (CBDs), with a single back-end (the Modelverse), and only a single implementation language (Python). Nonetheless, we believe that each of these is representative, and our approach does not depend on any of these in particular.

For **reliability**, we note that we depend on the familiarity of the researchers with the used tools. As we have used our own prototype tools, we knew all details relevant to the application of our approach. Lack of documentation about these tools might hinder other researchers from implementing the same functionality in this tool. Another threat to reliability is the small amount of experiments that were conducted.

6.3.4 Discussion

We briefly present three directions in which our work is currently still limited, but can be further extended: performance, GUI interaction and concrete syntax definitions.

Performance

Performance has not been discussed up to now, as it is not one of the concerns that we want to tackle. Nonetheless, concrete syntax can only be deemed usable if it is also sufficiently efficient to use: model perceptualization and comprehension require a relatively low latency, as otherwise the interface does not seem responsive, leading to user frustration. Model transformations are the crucial factor in our approach: benchmarking our approach would actually be benchmarking the underlying model transformation engine. Many model transformation optimizations have been discussed in the literature, such as incrementality [282], distributed queries [274], or scope discovery [150]. Our approach itself is independent of the underlying model transformation algorithm.

GUI Interaction

Up to now, the behaviour of the front-end was considered as a black box. While we did term its operations as *rendering* and *recognition*, nothing is said about how this happens. Many differences are possible here as well, which can ideally be domain-specific. For example, what operations does a modeller have to do to delete an element? Must an element be left clicked and then the *delete* key pressed, or is there a button to do this? Depending on the domain, any of these modes of interaction might be more natural to the user.

The behaviour of the GUI, and in particular its interaction with the user, should ideally also be explicitly modelled, similar to the concrete syntax. This timed, reactive, and possibly concurrent behaviour is best described by a specialized formalism, such as SCCD [298].

Concrete Syntax Definition

While our proposed framework offers a lot of flexibility to language engineers, defining a concrete syntax mapping is not as easy as an icon definition. To increase usability, we propose an additional language, the MM_{CS}, which is a language for concrete syntax definitions. A concrete syntax definition is a DSL for the definition of concrete syntax. An example is an icon definition language. Instances in this language, termed M_{CS}, can be used to generate the perceptualization and comprehension model transformation. So

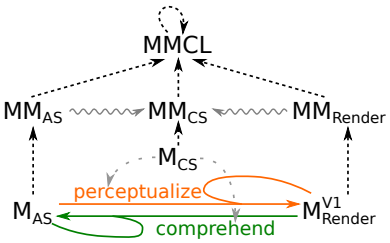


Figure 6.43: Concrete Syntax definition to automatically generate the perceptualization and comprehension operations.

while we use the full-blown infrastructure, it becomes possible to use a similar workflow as before, if so desired. This is shown in Figure 6.43, where we show that both model transformations are generated from M_{CS}. MM_{CS} is also tightly related to both MM_{AS} and MM_{Render}, as it uses concepts from both. Again, we are not restricted to a single MM_{CS}, as it is possible to define and use several, all of which define DSLs for the domain of concrete syntax definitions.

6.3.5 Related Work

Most (visual) modelling environments support customizing the concrete syntax of modelling languages. We consider a number of representative examples and explore to which extent they support the features listed in the previous subsections. Without exception, these tools hardcode MM_{Render}, meaning that even when they offer some dimensions of flexibility, it is constrained to a specific type of perceptualization.

AToMPM [273] is a graphical meta-modelling environment, implemented in Javascript/SVG. It allows language engineers to develop their languages' abstract syntax using a class-diagram language. For the concrete syntax, an icon definition language is provided. The language engineer has to create an icon for each class, and a link for each association. A class' icon and an association's link define the graphical appearance of the instances of that class or association; it can consist of several graphical primitives such as rectangles, circles, and lines. The graphical primitives have a number of attributes, such as *colour*, *size*, *font* (for text), etc. The value of these concrete syntax attributes can depend on the value of abstract syntax attributes: this can be defined in a *mapper*. Conversely, changes on the concrete syntax (e.g., dragging an icon) can be *parsed*, which results in changes to the value of the abstract syntax attributes. AToMPM is restricted to one-to-many perceptualization. Multiple concrete syntaxes can be defined for the same abstract syntax definition; the front-end allows to switch between different renderings of the same abstract syntax model. Due to AToMPM's client-server architecture, an alternative front-end could be developed using a different platform. Layout algorithms are not supported.

AToM^3 [86], the predecessor of AToMPM, is implemented in Python/Tkinter. Model storage and visualization are tightly coupled. Similar to AToMPM, visualization is defined using an icon editor, though only one concrete syntax definition is supported for each language, as they are tightly interwoven. No comprehension from concrete to abstract syntax is supported and perceptualization is limited to displaying the value of an attribute in a text field. The language engineer can, however, code actions that are triggered by

events, such as editing an object, moving it, selecting it, etc. These scripts can access both the abstract syntax and concrete syntax (Python) objects, though they are not governed by well-formedness rules: invalid configurations can be reached. Some layout algorithms are provided, such as circle layout and spring layout, though all of them are generic; domain-specific layout algorithms are not supported. For AToM^3, a multi-view component was previously introduced [85], though this was mostly focused on the abstract syntax and associated semantics.

MetaEdit+ [154] is a commercial domain-specific meta-modelling environment. To define the abstract syntax of a language, a metamodel is created in the feature-rich GOPPRR (Graph-Object-Property-Port-Role-Relationship) language. A symbol editor allows to customize the concrete syntax of the language; again, each class is given a graphical representation. Mapping is limited to text elements, whose value can be defined based on the abstract syntax of the model, and visibility of graphical elements, based on a condition on the abstract syntax of the model. Custom layout algorithms nor comprehension are not supported. While MetaEdit+ is a commercial, proprietary tool, it does implement a SOAP API with which external tools can query and modify the models stored in the tool. No access is given to the graphical info of the models. Therefore, it is impossible to implement a minimal user interface with MetaEdit+ as a back-end, unless perceptualization is implemented from scratch.

A number of frameworks exist that allow language engineers to create graphical user interfaces in Eclipse EMF[2]. GMF [3] allows the generation of a modelling tool from a concrete syntax definition, a perceptualization and a tool definition, which are all explicitly modelled. Users can generate an editor as an Eclipse plug-in or as a Rich Client Platform (RCP) application. Reusing existing libraries, however, is not as straightforward. Sirius builds on GMF and aims to ease the development of modelling tools, while primarily focusing on multi-view modelling [187]. Multiple concrete syntaxes for the same abstract syntax are supported, for example by providing multiple viewpoints depending on the level of abstraction. Papyrus [121] is a tool for modelling UML or SysML diagrams. Focusing on such standards, the tool allows users to specify tailored concrete syntax for their UML profile. All these EMF approaches are based on the generation of a modelling tool.

In the domain of textual languages, abstract syntax and concrete syntax are usually defined together by means of a grammar. In this context, comprehension is equivalent to parsing. Any (general-purpose) text editor can be used as a front-end for free-hand editing. A parser is used to determine the text's conformance to the language. Nowadays, smart text editors are used to parse the text dynamically during editing, thereby supporting syntax highlighting, error detection, auto-completion, etc. Xtext is a framework that supports implementing textual DSLs and such smart editors [106]. A DSL is defined by an Xtext grammar, from which it is possible to parse an EMF-based abstract syntax tree by using a generated ANTLR parser. A textual environment can be generated, which includes syntax highlighting, error visualization, content-assist, folding, jump-to-declaration and reverse-reference lookup across multiple files. Xtext supports multiple front-end frameworks, such as Eclipse, IntelliJ, and web browser support, but the user is not expected to define support for his own framework. Xtext is defined for textual languages exclusively, unlike our approach.

[2] `https://www.eclipse.org/emf`
[3] `https://www.eclipse.org/modeling/gmp`

Textual concrete syntax definition for DSLs is also supported in MetaDepth, based on ANTLR [84]. In MetaDepth, concrete syntax and abstract syntax definition are separated, unlike typical approaches for textual syntax. There is no dedicated support for a user interface; instead, an external general-purpose text editor must be used.

Similar to our approach is Monto [262], which addresses the problem of extending existing integrated development environments. But whereas their approach sticks to the same approach as before, trying to make plugins easier to define, our approach takes a radically different approach by modelling all aspects explicitly. In our approach, plugins disappear, and effectively become just new models in the tool, which are used to augment the behaviour of the tool. Projectional editing [326] is an alternative approach, where the abstract syntax, instead of the concrete syntax, is manipulated.

The overview of our approach bears similarity to the megamodel on parsing and unparsing [335], where 12 classes of artefacts were identified, along with a set of transformations between them. This overview is mostly oriented towards textual languages. In contrast, our approach covers different types of perceptualization: textual or graphical perceptualization is handled similarly in our approach. Related to this, our approach is capable of handling other perceptualization strategies as well, such as sonification, as long as there is an MM_{Render} and supporting front-end.

Summary

Current approaches to concrete syntax have several restrictions, in particular related to graphical concrete syntax. We identified five such restrictions which we address by presenting a Multi-Paradigm Modelling (MPM) approach. Our approach explicitly models concrete syntax mapping as a model transformation, which happens at the back-end, instead of in the front-end. This makes it applicable for all types of perceptualization and makes perceptualization available for different types of front-end. Changes to the rendered model are recognized in the front-end and comprehended by the server to update the abstract syntax model. We have shown the various dimensions of flexibility offered by this approach, and described our implementation in the Modelverse. The presented approach can furthermore be used in the context of collaboration, thereby facilitating the implementation and combination of model- and screen sharing [318].

6.4 Modelverse Debugging

Finally we consider the third type of user: the Modelverse developer. Recall that one of the responsibilities of the Modelverse developer was to create a bug-free environment for the other users to use. Having such a bug-free environment naturally raises the need for debugging the Modelverse, thereby finding the source of bugs. Debugging the Modelverse is hard, however, due to the complex nature of the tool. Indeed, the Modelverse was constructed using MPM techniques because of its complex and distributed nature. While debugging can be done at the level of the generated code, this is not ideal: the code is at the wrong level of abstraction and most parts of the code are synthesized from models. The Modelverse is also difficult to debug using existing code debuggers, as it is a distributed application. This means that there are multiple processes, possibly at multiple machines, all interacting with one another. When debugging, however, these applications would all have to be debugged individually.

In this section, we consider this problem in general: how can complex and distributed systems be debugged efficiently. We apply it to the Modelverse, which is also the example, though this technique is applicable in general.

6.4.1 Motivation

As the Modelverse is to be executed on a shared-resource machine, its interleavings with other processes is non-deterministic. The use of a network renders it even less reproducible: network delays are non-deterministic by nature. While developing such systems, bugs can be hard to replicate, as they are often related to these interleavings. For example, bugs causing deadlocks might only occur in a very select number of situations, and cannot be reproduced with absolute certainty. Such bugs are termed "Heisenbugs" [127], and they represent the majority of bugs in distributed applications. As debugging is an invasive operation, due to instrumentation, behaviour likely changes during debugging.

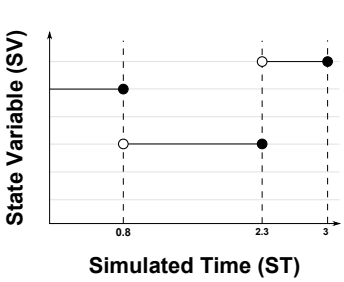
These problems are caused by the non-determinism of the underlying platform, being a timing issue. Debugging, and specifically instrumentation, affects timing as operations suddenly take longer to execute, or more operations have to be executed. Additionally, the operating system will perform different interleaving for different executions, thereby also altering timing. This results in potentially different behaviour, making debugging harder. To address these timing problems, we apply the same technique as with the performance model: we use simulated time by relying on a Parallel DEVS model.

6.4.2 Background: DEVS Debugging

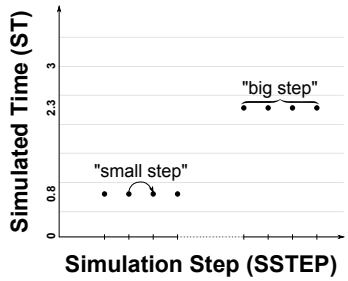
The use of Parallel DEVS for debugging naturally relies on the presence of a Parallel DEVS model debugger. Therefore, we elaborate on what debugging operations could be supported by such a debugger, and mention how such a debugger can be created. First, however, we present the notion of simulated time, which is of critical importance to simulation and debugging.

Time

The notion of *time* plays a prominent role in simulation [117, 124]. Simulated time differs from the wall-clock time: it is the internal clock of the simulator, instead of the time in the real world. In general, a simulator updates some state variable vector, which keeps track of the current simulation state each time increment. In contrast to wall-clock time, simulated time can be arbitrarily updated. This is shown visually in Figure 6.44a. The state is updated by some computations, transitioning from one consistent state to the other. All computation required between these two consistent states is called a "step". For each of these steps, a number of smaller computational steps may be involved. This is visualized in Figure 6.44b, where one "big step" is broken up into a number of "small steps". Note that the simulated time stays constant in between small steps, and only increases after a big step has been completed. While the state is guaranteed to be consistent after a big step has completed, this is not the case for small steps.



(a) Change of the state variable over time.



(b) Multiple steps executed on a single simulation time instance.

Figure 6.44: Simulation time and steps.

Debugging Operations

We now explore the different ways in which Parallel DEVS models can be debugged. We assume an operational view: the model is simulated through the use of an external simulator, instead of compilation to code. An appropriate model debugger should offer functionality similar to code debuggers: stepping, pausing, setting breakpoints, etc. Furthermore, formalism-specific debugging operations should be provided.

Pausing Pausing is a useful debugging operation, as it allows to interrupt a running program or simulation and inspect the current state of the system. A pause can either be manually requested by a user, or triggered automatically as a result of a breakpoint. The breakpoint specifies a condition on the runtime state, which, as soon as it evaluates to true, pauses the simulation.

State Inspection and Manipulation The state of a Parallel DEVS system is the combination of the states of its components, with each atomic DEVS models in exactly one state at the same time. Upon pausing the simulation, the user might observe that the model is in an unexpected state. It might then be useful to force changes to the system state, to observe the effects. We differentiate between two ways of manipulating the state: (1) *God events* alter state variables directly, but are invasive, and (2) *Event injection* sends an artificial event, thereby influencing the state variables indirectly.

Steps Based on the Parallel DEVS simulation algorithm, we distinguish three ways of stepping through the simulation of a Parallel DEVS model: “big step”, “small step”, and “step back”.

Big Step To see system behaviour on a fine-grained level, a user might step through the simulation to dynamically see the state evolving. We see this “big step” as the minimum amount of computation for bringing the system from one consistent state to the next.

Small Step For even more fine-grained control over the simulation, modellers might be interested in seeing the effects of the different phases in the simulation algorithm. We see a "small step" as the execution of one of the six phases of a big step: (1) compute imminent models, (2) generate output events, (3) route events, (4) find transitioning models, (5) perform transition function, and (6) compute time advance. Simulation is in an inconsistent state while "small stepping", and only becomes consistent if the big step is terminated.

Step Back Stepping back in time causes the simulation to revert to the last consistent state (i.e., revert the last big step). This is often termed "omniscient debugging", and it generally has a high impact on simulation performance. Nonetheless, it has become more popular recently and has often been praised for the additional insights in the model it helps reveal.

Tracing The previous techniques were concerned with "live debugging", which debugs the model during simulation. While this is useful, it is sometimes necessary to get a full overview of the system's execution history after simulation has finished. We call this *post-mortem* debugging.

Debugger Creation

In the context of this work, we created a visual modelling, simulation, and debugging environment for Parallel DEVS, based on the PythonPDEVS [308] simulator. To manage the inherent complexity, we de- and reconstruct the PythonPDEVS simulator [308] and add debugging support to its modal part (which is explicitly modelled in the Statecharts [133] formalism) of the simulator. This is based on our previous work [293], where the de- and reconstruction approach was introduced for instrumenting model simulators. Opportunities for Parallel DEVS debugging were first explored in [296]. We combine the simulator with the visual modelling tool AToMPM [273], allowing for the visual interactive control of DEVS model simulations. Although we chose two research tools with which we are familiar, the same technique can be applied to other environments and simulators with similar capabilities.

The process of de- and reconstruction is not presented in detail, as it does not contribute to the final result: how to debug Parallel DEVS models. The interested reader is referred to [293, 294, 296, 300]. In essence, the PythonPDEVS simulator's modal behaviour is modelled using Statecharts, as shown in Figure 6.45. Generic debugging operations are then merged with this model, to offer debugging operations, as shown in Figure 6.46..

Efficient Omniscient Debugging

With omniscient debugging, modellers can jump back to arbitrary points in past simulated time. It has recently received much attention in the programming language domain (*e.g.*, in GDB [120]), as it can overcome problems commonly associated with breakpoint-based debugging [178, 222]. The most prominent being that erroneous behaviour probably occured before the breakpoint was triggered, for which most information is lost already. Despite the many advantages of omniscient debugging, implementing it efficiently is

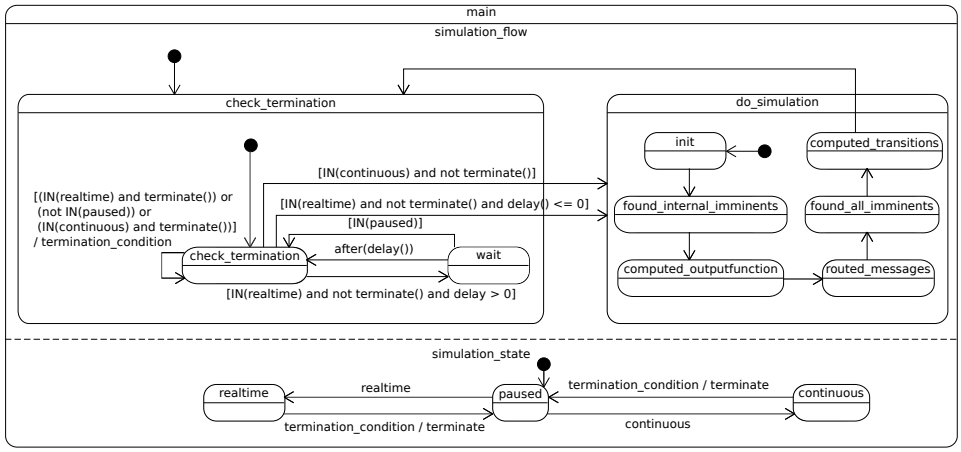


Figure 6.45: PythonPDEVS Statechart.

challenging: slowdowns of a factor 100 or more are noted in the literature [181]. We consider two omniscient debugging operations: taking a single simulation step back, or jumping to an arbitrary point in simulated time.

We transpose techniques from the domain of optimistic synchronization, and in particular Time Warp [146], to omniscient debugging. Minor modifications are required, as omniscient debugging has different priorities: rollbacks can take some time and a complete history of the model must remain available. Contrary to other omniscient debugging approaches, this algorithm is lossless and has a low overhead in both time and space.

Problems with Omniscient Debugging Despite omniscient debugging's advantages, there are severe performance limitations, making it unsuited for large-scale models. First, omniscient debugging is plagued with memory issues [50, 71, 223]: storing the complete simulation trace eventually leads to memory exhaustion, as trace size only increases. Most omniscient debuggers tackle this problem in a lossy way: using a time window (only store the last x states) or using partial states (only store the state for several models). Second, omniscient debuggers have low (forward) simulation performance [181, 222]. The primary overhead is in serializing and storing model states after each transition.

For our initial Parallel DEVS debugger, Figure 6.47 shows the difference in (forward) simulation time between turning omniscient debugging on and off, dependent on the size of the state. We see that execution time increases as the state history increases, due to the serialization overhead of state saving becoming the bottleneck. The increase is linear, as the serialization routine used has linear complexity in terms of the state size. This overhead is always present when the option for omniscient debugging is provided, even when it is never actually used. The sporadic use of omniscient debugging, therefore, does not warant the significant overhead on the more frequent forward simulation operations.

Link to Time Warp As the cause of this dual performance problem lies with the state saving that is performed, we look at this algorithm in detail. The core problem is: "how

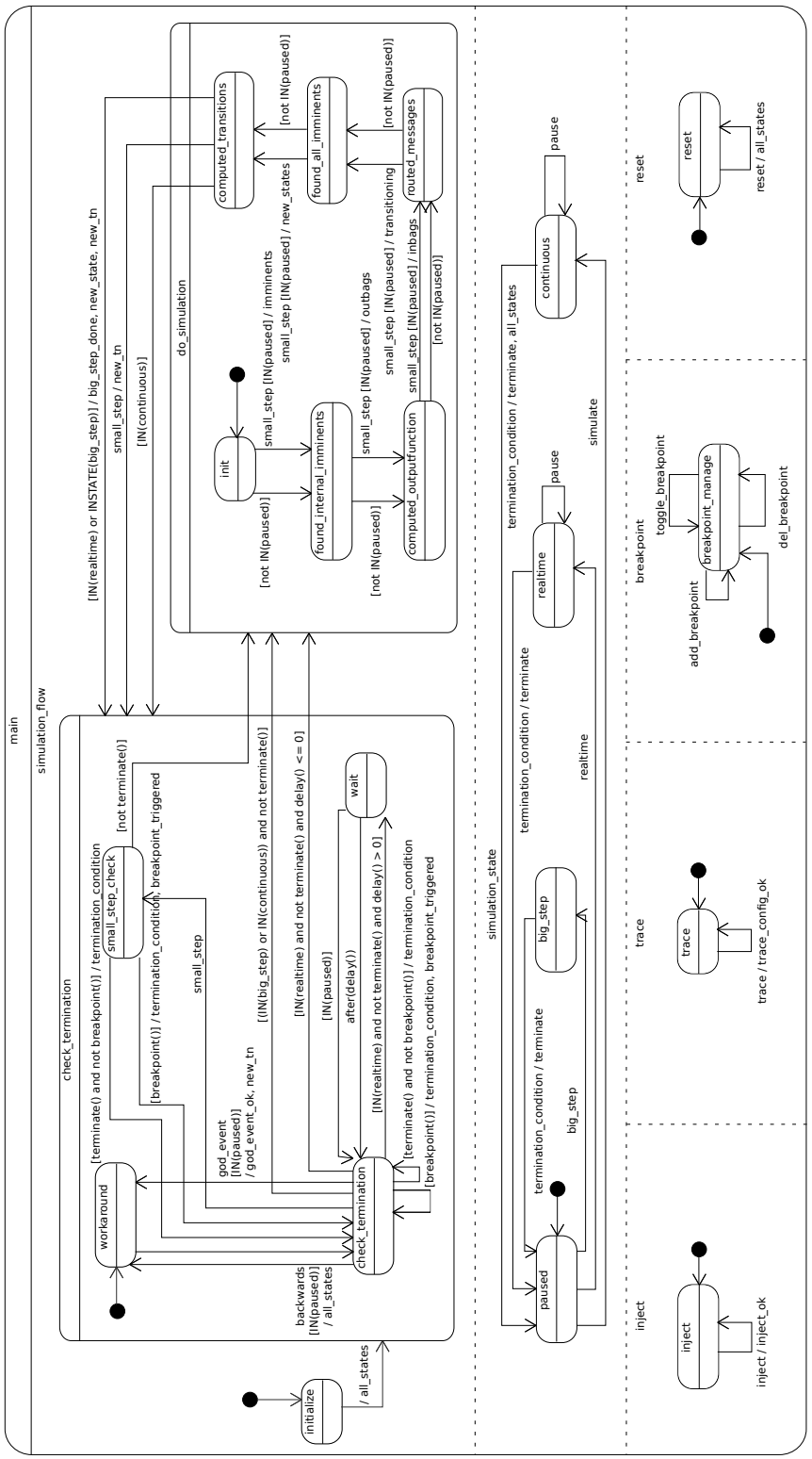


Figure 6.46: PythonPDEVS Statechart augmented with debugging functionality.

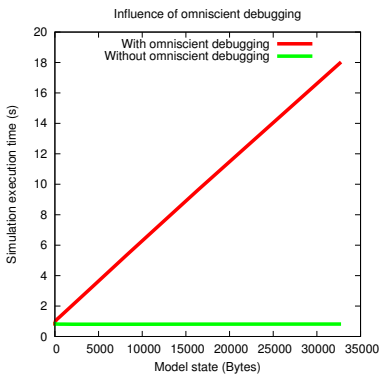


Figure 6.47: Overhead of omniscient debugging in forward simulation.

to jump back to an arbitrary point in history?". This is the same problem encountered in optimistic synchronization, and in particular Time Warp.

In the context of Time Warp, several algorithms were created [68, 224, 236], with varying degrees of stored data. *Full State Saving* stores a complete model snapshot at each transition, as discussed before. *Copy State Saving* stores a snapshot of a specific model that changes its state, as discussed before. *Incremental State Saving* stores only the difference between two subsequent states, in the form of a reverse operation. During a rollback, all state changes need to be undone in reverse order. This makes the length of the rollback influence the time taken for the rollback. *Periodic State Saving* will, instead of storing the model state after every transition, only store the state periodically. During a rollback, we select the closest state before the requested time, and simulate from then on. This assumes determinism in the simulation algorithm, as otherwise it is not guaranteed that the same choices are made.

There are different non-functional requirements between Time Warp and omniscient debugging. First, Time Warp solves the memory problem by using a window-based approach. Contrary to omniscient debugging, however, optimistic synchronization can place a lower bound on the states that will be accessed, using the Global Virtual Time (GVT), allowing it to use a window. This is not the case with omniscient debugging, as we cannot know what state the user wants to go back to. Second, rollbacks occur often in Time Warp, and need to be processed fast to prevent cascading rollbacks [117]. This is again not the case with omniscient debugging, where backwards steps happen only rarely and performance is less of an issue.

For Time Warp, the main disadvantage of periodic state saving is that it requires forward simulation for each backward step. This makes a backward step take longer than a forward step (as one includes the other). But although this is a substantial problem for Time Warp, omniscient debugging is used interactively and only rarely. So wheras a latency of 0.1 seconds is too much for Time Warp, even latencies up to half a second might be tolerable during omniscient debugging. So since periodic state saving's disadvantages are minimal for omniscient debugging, we consider this algorithm for our implementation of omniscient debugging.

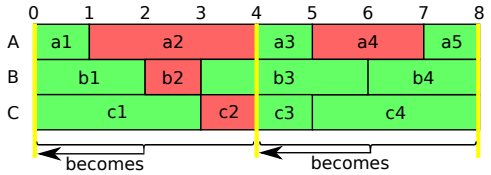


Figure 6.48: Overview of periodic state saving approach. Green states (light) are stored, red (dark) states are not. Yellow lines indicate a point at which a snapshot is made.

Periodic State Saving for Omniscient Debugging Our algorithm [306] is based on Periodic State Saving: instead of storing the state of models at transition-time (as in copy state saving), we store the full simulation state after a fixed interval. This does not influence the forward simulation algorithm at all, as storing the model state happens independent of forward simulation. For backward steps, we search the most recently saved simulation state, revert to it, and forward simulate from there up to the requested time. Users can configure the interval, thus influencing performance.

The checkpointing interval is defined in wall-clock time (i.e., real world time), instead of simulation time (i.e., internal clock of the simulator). While simulation time provides deterministic points in the simulation where snapshots are made, using the wall-clock time takes into account a possibly changing simulation pace. Time efficiency, and latency, is expressed in wall-clock time, as that is the actual time that the modeller will have to wait for operations. Defining the interval in number of events executed would also be possible, but has similar disadvantages as basing it on simulation time. The time taken for the forward simulation phase is bounded by the snapshot interval: with snapshots every x seconds (of forward simulation), a rollback never requires more than x seconds of forward simulation to reach the desired state, as otherwise another snapshot would have been closer.

We also allow users to configure the maximally allowed memory use. When simulation uses more memory, the oldest full model snapshots are compressed and persisted to disk. Since these old snapshots are very unlikely to be necessary, and responsive performance is all that we require, there is no significant disadvantage to disk storage. This way, the full disk space becomes available for use by omniscient debugging, without any noticable performance impact.

An overview of the approach is shown in Figure 6.48, where only three snapshots are made. When rolling back, the latest snapshot is selected and simulation is restarted from there on, until the requested time is reached. For example, when rolling back to time 6, state $a4$ is missing, making us roll back to time 4, where a snapshot was previously made. From here, the transition function resulting in $a4$ is executed again, to yield the total state at time 6, as requested.

Nonetheless, main memory is still limited and will eventually fill up. Therefore, old state snapshots can be persisted to disk when additional main memory is needed. When an older state snapshot is required, it is just read out from disk. While disk accesses have much higher latency, the additional delay is only in the order of several milliseconds, and is only induced once, since a single snapshot is made for the complete state. Including access and transfer times, reading data still feels interactive, as it only adds milliseconds to the total time of a rollback. The cost of the forward simulation phase is many times higher. Writing

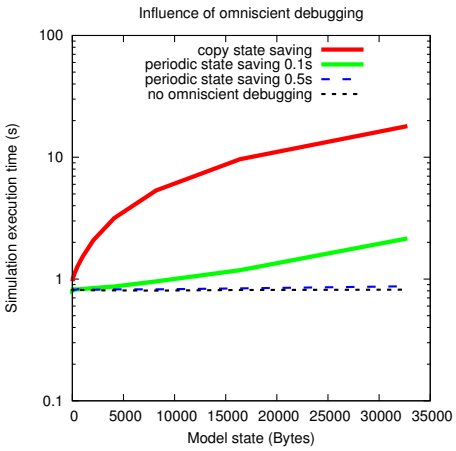


Figure 6.49: Overhead of omniscient debugging in function of state size (logaritmic scale).

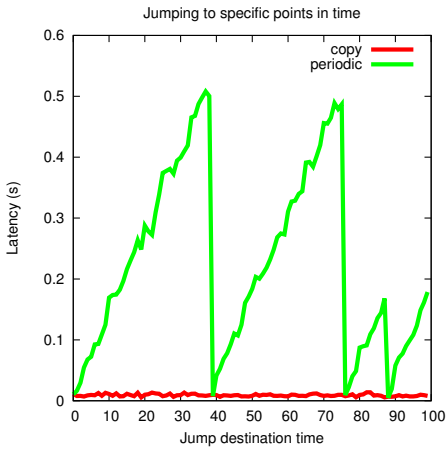


Figure 6.50: Jump latency for copy and periodic state saving.

out to disk is slower as well, but happens asynchronously in most operating systems, thereby mostly avoiding the problem. To further minimize the amount of data that is stored and has to be written to disk, snapshots can first be compressed using existing general purpose compression algorithms. The reduced time to write out the compressed snapshot is often already sufficient to make up for the overhead of compression and decompression.

We evaluate this approach and its configurability in two dimensions: time and memory.

Time The user is free to configure the interval between consecutive snapshots. Setting a *longer interval* between two snapshots results in: (1) *lower memory consumption*, since snapshots are saved less frequently; (2) *faster forward simulation*, since less serialization pauses occur; (3) *slower backward simulation*, since less states are saved, requiring more forward simulation to reach the requested rollback time.

Figure 6.49 shows forward simulation results for periodic state saving, compared to copy state saving. Two snapshot intervals are considered: 0.1 seconds and 0.5 seconds. In both cases, forward simulation overhead is much less than usual. For 0.5 seconds, the overhead is almost neglible. Figure 6.50 shows backward simulation results: how long it takes to jump to a specific point in time from simulation time 100. In this case, periodic state saving has an interval of 0.5 seconds, and we indeed see that the maximum time for a backward simulation step is bounded by 0.5 seconds. As such, a 0.5 seconds interval has an almost neglible impact on forward simulation, while backward simulation is impacted, though still feels responsive to users.

Memory The user is free to configure the threshold as to when old snapshots are written out to disk. Storing *more in main memory* results in: (1) *higher memory consumption*, since more snapshots are stored in memory; (2) *faster forward simulation*, since less writes to

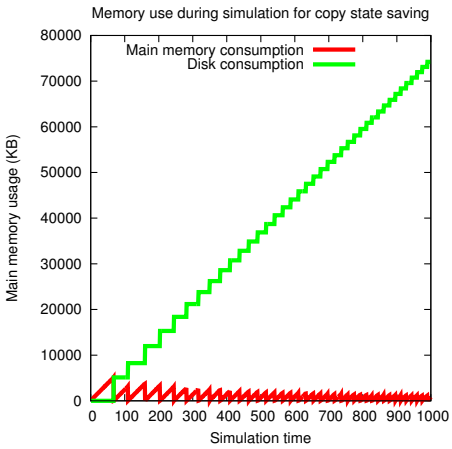


Figure 6.51: Memory use of copy state saving.

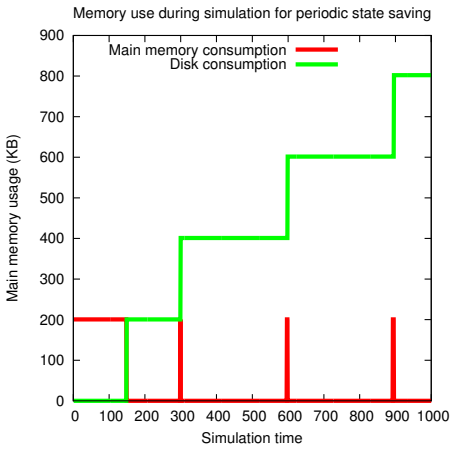


Figure 6.52: Memory use of periodic state saving.

disk are required; (3) *faster backward simulation*, since the snapshot is more likely to be in main memory.

Figure 6.51 presents the memory used for copy state saving, as is the naive approach. Memory use increases rapidly, as all intermediate states are persisted. At this rate, it is likely that the disk becomes too slow to write out all the date, thereby even having an impact on forward simulation performance. Compare this to Figure 6.52, which presents results for the same simulation but now using periodic state saving. In this case, only three snapshots are stored, which reduces the required amount of memory by a factor 100 (800 KB instead of 80 MB). This amount of data is also sent to the disk in less requests, making the latency less noticeable.

Long-running Simulations Even now, our approach cannot handle arbitrarily long running simulations: just like main memory, disk space eventually runs out. Despite optimizations to increase the capacity of our storage media, such as file compression, this only delays the point where memory inevitably runs out. Infinitely running simulations can however still be supported at the cost of increased latency for omniscient debugging operations. By pruning away intermediate snapshots persisted to disk, we gain more storage space for future snapshots. This comes at a cost, as each snapshot was there to guarantee the initially defined latency. Whereas our approach still works even with less snapshots, latency increases, but remains bounded. For example, when removing every other snapshot, average latency doubles, though memory consumption halves. This can keep going on, though latency doubles each time. Nonetheless, the time it takes to reach the requested state stays bounded. This differs from a window-based approach in that we remain lossless. Our approach is also guaranteed to never be slower than restarting the simulation completely, as a restart is just the worst case situation, in which there is no closer snapshot available.

Related Work

Related work can be divided in three main dimensions. First, model debugging is a more general approach to the debugging of DSLs, which could therefore also be applied to DEVS, given that DEVS can be considered a DSL. Second, some approaches exist to DEVS debugging, though they mostly rely on code-based approaches and are not centered around debugging. Third, efficient omniscient debugging techniques are used in various domains, such as code debugging and model debugging.

Model Debugging We believe debugging support for modelling and simulation has to be provided at the most appropriate level of abstraction (*i.e.*, using the abstractions of the formalism, instead of relying on low-level code). In [12], the authors explore requirements for modelling and simulations tools that support verification, validation and testing. One of those is the need to present concepts at the domain-specific level. Debuggers for some other formalisms already exist. In [205], Mustafiz and Vangheluwe construct a debugging environment for Statecharts, by instrumenting the Statecharts model with appropriate transitions. We take inspiration from their approach, but generalize their technique to apply it to other formalisms as well. A debugger for Modelica was developed in [220]. Model transformation debugging, in particular omniscient debugging techniques, were explored by Corley in [69].

Methods specific to the debugging of Domain-Specific Languages (DSLs) have also been researched. In [189], Mannadiar and Vangheluwe address the need for debugging models in DSLs and propose a mapping of code debugging concepts to model-based design. A notable work that attempt to generalize techniques for adding debugging support to DSLs is the Moldable Debugger [63], a reusable framework for developing debuggers for DSLs. It allows to implement a set of debugging operations such as stepping, state querying, and visualization at the most appropriate (domain-specific) level of abstraction. In [50], the authors describe a partly generic debugger that can be extended with domain-specific trace management functions. They allow the definition of a set of debugging operations that traverse, query, and manage these execution traces. We take inspiration from their work to map code debugging operations onto domain-specific debugging operations (in our case, specific to Parallel DEVS).

We contribute to this emerging field by developing a debugger for Parallel DEVS. In particular, we define a set of useful debugging operations, based on existing code debugging techniques as well as simulation-specific operations. We add these operations to the existing PythonPDEVS [308] simulation kernel using a generic technique which we call the de- and reconstruction of the simulation kernel. We couple it to a (basic) interactive visual user interface. In [188], a set of visual interfaces for working with Classic DEVS models is introduced. The goal is to convey the meaning of the models intuitively, by providing a visual notation for the design, but also the execution trace of the simulation. This allows model debugging through the visual inspection of this trace. Closely related is the work by Kemper [155], who presents a method for debugging stochastic models based on a visualization of their trace, from which irregular patterns can be discerned. Our work is complementary, as we focus on dynamic debugging techniques using different types of steps, execution modes, breakpoints, etc. Our visual interface displays limited information, and, because of our modular architecture, can be replaced by a more advanced one.

	ADEVS	DEVS Suite	MS4 Me	VLE	X-S-Y	PyPDEVS
Pause	C	Y	Y	N	Y	Y
(Scaled) Realtime	C	Y	Y	N	Y	Y
Big Step	Y	Y	Y	N	Y	Y
Small Step	C	N	N	N	N	Y
Termination condition	Y	N	N	N	N	Y
Breakpoints	N	N	N	N	N	Y
Event Injection	C	Y	Y	N	Y	Y
State Changes	N	N	N	N	N	Y
State Visualisation	C	C	C	N	C	Y
Event Visualisation	N	Y	Y	N	N	Y
Tracing	C	Y	Y	Y	Y	Y
Model Visualisation	N	Y	Y	Y	N	Y
Reset	N	Y	Y	C	Y	Y
Step back	N	N	N	N	N	Y

Table 6.1: A comparison with several other DEVS debugging tools.

DEVS Debugging There are a number of commercial and research DEVS modelling and simulation tools that provide some form of debugging. Table 6.1 compares five of them with our approach, implemented in PythonPDEVS. For each function, we list whether or not the tool implements it (**Y** for yes, **N** for no), or if there is some preliminary or partial support (**C**).

Efficient Omniscient Debugging Omniscient debugging was first explored in the context of General-Purpose Languages (GPLs). The main difference is their non-determinism: user input and I/O events have to be stored in order to correctly step back in time. Pothier *et al.* use events to monitor the running execution [223]. These events are stored in a database, which can be distributed to further increase performance. Boothe explores techniques for efficient bidirectional debugging of GPL programs [49]. These techniques are based on event traces (to deal with non-determinism) and snapshotting (for increased performance). Older snapshots are progressively removed as the program is executed (for memory efficiency). Ultimately, however, memory runs out, as removing events from the trace is not an option. The only solution is to become lossy: dropping events from the trace, or limiting the number of backward steps the modeller can make (*i.e.*, a window). If it turns out that the user is interested in these events after all, the program needs to be reset and executed again. Some approaches, such as reverse computation [340], partly avoid the problem of memory consumption. But while they avoid one problem, reverse computation is computationally more intensive for long jumps, and not even always possible. Engblom presents an overview of different techniques for omniscient debugging of GPLs [103]. Recently, support for omniscient debugging has also been included in mainstream tools, such as GDB.

Omniscient debugging techniques have been explored in the context of modelling languages. Corley *et al.* have implemented omniscient debugging for model transformations and analysed its efficiency [70, 71]. Since model transformations are non-deterministic, their

implementation logs each change at the end of a transformation step. By inverting these changes, users step back to previous states. Overhead is limited, as it is incremental in nature, though it eventually runs out of memory unless old events are dropped or persisted to disk. Neither of these handles long running simulations losslessly: disk space can still run out. Time can also present a minor problem during a rollback, as it is linear in the length of the rollback: history unrolls step by step.

In contrast to model transformations, Parallel DEVS is a deterministic formalism, meaning that we can remove arbitrary intermediate states: they can always be computed again. In [50], the authors explore the debugging of domain-specific languages. Their approach is based on the saving of a trace during execution, which can be explored backwards and forwards by the modeller. They allow for domain-specific languages to specialize the generic trace algorithm, to gain efficiency in space and time.

The literature on omniscient model debugging is rather sparse and does not include many lossless optimizations or solutions for the memory management problem. Existing approaches mostly focus on code debugging, or rely on lossy techniques. And while we agree that lossy techniques are sometimes necessary (*i.e.*, for non-deterministic formalisms), we can make additional optimizations in the case of Parallel DEVS. The literature on optimistic synchronization protocols [115, 117], however, has extensive work on optimizing rollbacks in a deterministic and lossless way. Many variations to, and evaluations of, state saving algorithm exist [68, 224, 236]. We have based ourselves on these algorithms to define a lossless, time- and space-conscious omniscient debugging algorithm.

Summary

We created an advanced debugging and experimentation environment for Parallel DEVS models, offering the user a level of control unmatched by any of the state-of-the-art tools. The supported operations are similar to those of traditional code debugging tools, though some operations specific to DEVS were added. Simulation can be paused, resumed, and stepped through, breakpoints can be set, events can be injected, and the state can be modified directly. Our approach adds features that are not found in any other tool, such as god events and stepping back in time. To tackle the complexity of constructing this debugger, we excplitly modelled the modal part of the simulator. Apart from offering these advanced features, stepping back in time was implemented efficiently, making it applicable to large-scale models as well. For this, existing Time Warp state saving algorithms were reused and adapted in the context of omniscient debugging. A performance evaluation was given in terms of forward simulation time, backward simulation time, and memory consumption.

6.4.3 Model

To debug the Modelverse using a Parallel DEVS model, we of course first need this model. Luckily, this is exactly the same model as was presented in Section 5.12. Indeed, this was also a DEVS model and it included non-model code as well. Therefore it becomes possible to run the simulation instead of executing the application.

6.4.4 Evaluation

While the original application could be debugged using commonplace code debuggers, we claimed that this is not ideal for distributed and performance-critical applications, such as the Modelverse. Code debuggers often rely on instrumentation of the source code or binary, thereby altering the executed code. While instrumentation does not modify the semantics of the code directly, it might influence the semantics by altering the execution time of code fragments. Attempts have been made in the literature to minimize instrumentation overhead, though advanced features, such as omniscient debugging, generally always have a noticable impact. We consider two shortcomings of code debuggers in our context: lack of deterministic debugging and advanced debugging features. We elaborate on each problem, and mention how DEVS modelling and simulation addresses this.

Deterministic Debugging

The first problem is the lack of deterministic debugging. When debugging, the code is almost always instrumented in one way or the other. The simplest way of debugging, adding print statements, obviously changes the source code. Even worse, it also changes the timing behaviour: the print statement has to be executed, causing all operations after it to start later, possibly changing interleavings. All approaches monitor the application in one way or the other, resulting in ever so slight changes in execution behaviour. This makes it difficult to replicate bugs while using a debugger, or even on a second try. As a result, the same execution might have to be replicated many times, just to make sure that the bug manifests itself. But even when the bug can be reproduced frequently, the patches that are applied to further track down the bug (e.g., more print statements), or even to fix the bug (e.g., change some algorithm), can merely mask away the bug, instead of actually solving it. When non-determinism is involved, we often cannot be certain whether a bug just no longer manifests itself, or was completely fixed.

With the DEVS model, we have previously mentioned that we achieved determinism for performance benchmarks. But this determinism is also present during debugging: when nothing is altered in the simulated time, any change to the algorithms has no effect whatsoever on the simulation results. Therefore, print statements can be liberally added, and we can be sure that the bug is reproducible.

Advanced Debugging Features

The second problem is related to debugging features. While code debuggers have lots of features, some intrusive features, such as omniscient (reversible) debugging, are only selectively enabled, as they have a huge performance impact (e.g., up to $50,000$x for omniscient debugging in GDB) [4]. Enabling such features aggravates non-determinism and makes it even more difficult to replicate the bug. Nonetheless, even with a code debugger on our DEVS simulation, the huge performance penalty makes it difficult to use, and it then works at the wrong level of abstraction.

With the DEVS model, however, the level of abstraction can be raised by using a DEVS debugger. While they both debug the same application, they do so at a different level

[4] https://softwareengineering.stackexchange.com/questions/181527/why-is-reverse-debugging-rarely-used

of abstraction: code debuggers debug each line of code, and for omniscient debugging, this means that each line of code can be "reverted". DEVS debuggers, however, debug transitions of the DEVS model, which are composed of hundreds or thousands of lines of code. Naturally, the performance impact is decreased significantly. Additionally, advanced debugging features that are already implemented for the specified DEVS simulator can be used as-is, such as all the features implemented for PythonPDEVS [300, 306]. This opens up many new dimensions to the debugging of our application, while limiting the performance impact [223].

6.4.5 Related Work

Traditional debugging approaches have troubles with distributed systems, as they have to coordinate different systems and platforms. Some code debuggers do exist, though they have a limited set of functionality in a distributed setting. Even for non-distributed programs, code debuggers suffer from the difficulty of reproducing a bug, often called intermittent bugs, or "Heisenbugs" [127]. Heisenbugs are bugs which "go away when you look at them", and could elude programmers for years of execution [127]. In non-distributed programs, Heisenbugs are much less frequent, as they are often caused by hardware faults. Much work has been spent on decreasing the overhead of instrumentation, such as using dedicated hardware [270] or dynamic instrumentation [344]. Nonetheless, an overhead still exists, thereby altering the behaviour between runs. Related to our work, full system simulation [11] can be used to completely simulate the hardware on which we are executing. Compared to our approach, full system simulation is more general, as it allows all types of applications to be executed. Nonetheless, due to the very low level of abstraction, that of the CPU instructions, debugging is made difficult again, as we are debugging using low-level concepts, instead of programming language constructs or modelling concepts. As such, this approach only seems useful when debugging parts of the operating system. Additionally, performance is reduced even further, as execution is estimated to take about 50-200 times as long [11]. No mention is made about using this approach for distributed applications. For the previously mentioned DEVS models, no mention was made about debugging using the DEVS model. A distributed version of PythonPDEVS [309] was modelled using DEVS, and was used for debugging [303]. While no extensive DEVS debugger existed back then, the deterministic reproducibility of the bug was immensly helpful in debugging problems with the distributed synchronization algorithms.

Summary

During the development of complex and distributed applications, such as the Modelverse, problems arise that hinder debugging with today's code debuggers. Instead we debugged the Modelverse's DEVS model, thereby offering full control over time: wall clock time and simulated time are effectively split. This split gives us the ability to debug the application without interfering with the delicate timing mechanics underlying the system, and due to the higher level of abstraction, we can reuse advanced debugging operations, such as omniscient debugging, without excessive overhead. While these results were obtained in the context of our prototype MPM tool, we believe it to be applicable to many other complex, distributed applications. Advantages of using DEVS modelling and simulation for tool development are relatively easily achievable, given that much code can be reused, as was the case for our tool.

Summary

We started this chapter by linking back to the original requirements in the context of the Power Window case study. For each of these requirements, we elaborated on how the Modelverse provided support and was thereby able to correctly execute the case study. The Modelverse can therefore be called a Multi-Paradigm Modelling tool. Subsequently, we investigated the applicability of the Modelverse in future research by providing one contribution for each type of Modelverse user that we originally identified. This was possible thanks to the design of the Modelverse, which provides support for MPM and is built using MPM techniques. We presented live modelling for the modeller, as a way of dynamically altering the model during execution, thereby increasing model comprehension and insight. We presented a flexible form of concrete syntax for the language engineer, as a way of defining more complex and non-standard (e.g., not icon-based) concrete syntax for their languages. We presented DEVS-based debugging for the Modelverse developer, as a way of deterministic and configurable debugging of a complex, distributed system.

Chapter 7

Conclusions

This thesis set out to contribute to the state-of-the-art in Multi-Paradigm Modelling (MPM). On the one hand, we provided the necessary foundations and tool support for future research in the domain. On the other hand, we developed this tool using MPM techniques ourselves, providing a set of benefits over other tools.

We motivate this research by looking at the complexity of today's engineered systems. At run time, software controls hardware components in a feedback loop, the complete system has to interact (safely) with the environment, and often multiple such systems are connected over a network and have to cooperate to achieve a task. At design time, these runtime requirements often require multiple languages and tools to be combined, in order to create a single big system. To handle these problems, MPM proposes to explicitly model all relevant aspects of the system, using the most appropriate formalism(s), at the right level(s) of abstraction, while explicitly modelling the process. We made the assumption that MPM indeed provides the means of handling these problems, based on collected experiences from academia and industry. Despite the often claimed advantages of MPM, tool support was limited and often focussed exclusively on a few requirements related to MPM (e.g., language engineering, model activities, process modelling, multi-users). This is not surprising, as each of these aspects contitutes its own research domain, with its own community. No unified foundation for MPM existed, which incorporates all aspects of MPM, and is additionally usable for future extensions and applications in the domain.

There are three parts to this thesis: 1) we created the specification for a prototype tool, 2) implemented a prototype using MPM techniques, and 3) applied it for research in MPM.

Modelverse Specification

First, we created a specification for a tool with support for all aspects of MPM, termed the Modelverse, as described in Chapter 4.

Types of Users Our starting point was the different types of users that would use such a tool. We identified three: the modeller, the language engineer, and the Modelverse tool developer. The modeller creates correct models of the system with respect to some intended properties, ensuring that the models can be used as an abstraction of the intended system. The language engineer creates the necessary languages for the modeller, ensuring that these languages are as intuitive and usable by the modeller as possible. The Modelverse tool developer creates the necessary tooling for the other users, ensuring that they can use the tool according to the specification (e.g., bug-free, efficient, distributed).

Requirements In the end, ten requirements were distilled for a tool for MPM, based on the definition of MPM and the needs in typical MPM scenario's:

1. Domain-specific languages and models in these languages must be creatable.
2. Activities must be specifiable and executable in many different formalisms.
3. Process models can be created and enacted using previously defined models.
4. Multiple (distributed) users equally share computational resources.
5. Multiple external (proprietary) services must be able to connect.
6. Multiple interfaces must be supported, possibly using a different platform.
7. Models must be sharable between users and groups.
8. User access control regulates sharing of models between users and groups.
9. Links between models must be representable and can be manipulated using megamodels.
10. All tool components must be fully portable between platforms.

Architecture Based on these requirements, we proposed an architecture based on a client-server setup. The Modelverse Interface (MvI) is the interface to the Modelverse, or the client. It is responsible for making the Modelverse tool usable by the end-user. The Modelverse Kernel (MvK) is the client-facing compont of the Modelverse server. It is responsible for all computation, taking in the requests of the MvI and translating them to state changes. The Modelverse State (MvS) is the database back-end of the Modelverse server. It is responsible for effecting all state changes dictated by the MvK. For each part of the global architecture, the interface was described.

Modelverse Implementation

Second, we created this tool through the use of MPM techniques itself, as shown in Chapter 5. This was motivated by the claimed benefits of MPM when used for complex systems, which we assumed to be true. The Modelverse will indeed become a complex system due to the nature of the requirements mentioned before. For example, there will be a networking component and interaction with multiple (geographically distributed) users and (proprietary) tools. By adhering to the MPM approach, we explicitly modelled all relevant

aspects of such prototype tool. These aspects included some components that were easier and more efficient to develop using domain-specific models, such as the user interface, client API, networking components, and the task manager. For several other aspects, discussed next, more advantages came to light, thereby addressing problems encountered in current tools.

Conformance Relation By explicitly modelling the conformance relation, we can dynamically support multiple types of conformance in the Modelverse. This is necessary to address the different non-interoperable implementations of conformance found in today's (meta-)modelling tools, which hindered the use of model repositories. Additionally, this makes it possible to support multiple conformance relations for the same model, possibly each with their own semantics.

Physical Type Model By explicitly modelling the physical type model, it becomes possible to implement all model operations in the linguistic dimension, instead of the physical dimension, as is usually done. This is necessary to abstract away from the physical implementation, allowing for different physical implementations while maximally reusing model management operations. Additionally, this makes it possible to perform activity-based optimizations, where the tool optimizes its internal data representation based on domain-specific information.

FTG+PM Enactment By explicitly modelling FTG+PM enactment semantics by mapping it to an SCCD model, it becomes possible to support FTG+PM enactment on different platforms without the need for many complex implementations, in particular with relation to the concurrent behaviour. This is necessary because the implementation of enactment is non-trivial due to the required concurrency. Additionally, this makes the FTG+PM susceptible to novel analysis techniques that are applicable in the Statecharts domain.

Service Orchestration By explicitly modelling service orchestration in the SCCD formalism, it becomes possible to provide an explicitly modelled interface for a black-box service. This is necessary to combine non-modelled components in an explicitly modelled context, thereby leveraging the best of both worlds: the functionality of the black-box component, with the interface of a white-box component. Additionally, this can be combined with the explicitly modelled FTG+PM to achieve analyzability up to the level of the different services and their interaction.

Action Language By explicitly modelling the action language semantics through graph transformation rules, it becomes possible to automatically generate not only (always up-to-date) documentation, but also an interpreter. This is necessary, as the interpreter should be easily portable between different platforms, while guaranteeing that the exact same semantics is used. Additionally, the explicitly modelled abstract syntax graphs is easy to use as source or target of activities, making it possible to automatically manipulate such models.

Performance By explicitly modelling and simulating the performance, it becomes possible to deterministically assess performance in a variety of (hypothetical) scenario's. This is necessary to allow for deterministic benchmarks and what-if analysis in a variety of situations. Additionally, simulating the performance might be more efficient than actual execution, as many sources of overhead can be simulated as well, such as the network latency and bandwidth.

Modelverse Application

Third, we applied this prototype tool in the context of FTG+PM enactment for a power window case study, as often used in the MPM literature, thereby evaluating our tool in an MPM context. Additionally, we used this prototype tool for further research in the domain of MPM, thereby focussing on the three types of users we previously identified. For each of these users, we aid them in their responsibilities by offering new techniques and functionality. For the modeller, we support live modelling, where an executable model can be manipulated during execution, with the changes being taken into account in the current execution already. For the language engineer, we support more flexibility in the concrete syntax, allowing them to move away from the traditional icon-based concrete syntax. For the Modelverse tool developer, we support additional debugging functionality in a deterministic debugging context, by debugging the underlying DEVS model.

Live Modelling Live modelling allows modellers to alter the design model during the execution of that very same model. As such, modifications have a direct influence on the execution trace that is at that point being generated. While several such approaches already exist in the live programming domain, with some crossovers to the live modelling domain, these are mostly ad-hoc approaches. We proposed an explicitly modelled process which is generic for many different types of modelling languages, which we have evaluated for three representative languages: Finite State Automata, Discrete Time Causal Block Diagrams, and Continuous Time Causal Block Diagrams. This approach requires an MPM context due to its use of a structured process, containing several intermediate formalisms and (automated) activities between them. Additionally, this approach is applicable in an MPM context, as MPM naturally promotes the use of many domain-specific languages, for which we want to increase the set of available operations with a minimum amount of development effort.

Concrete Syntax Concrete syntax encompasses the different ways in which a model is presented towards the user. As such, making this presentation more flexible, for example by offering alternative ways of perceptualization (e.g., sound for a music sheet model), can make the model representation more natural, and thus more intuitive for the users. Other limitations on concrete syntax that we addressed, include strong coupling with the front-end, lack of domain-specific lay-out algorithms, lack of different concrete syntax views on the same model, and lack of complex notations (i.e., not a simple icon-based representation where one abstract syntax element is visualized by several concrete syntax elements). We proposed an explicitly modelled process for the perceptualization and rendering of abstract syntax models, which is agnostic to the actual rendering format. This approach requires an MPM context due to its use of a structured process, containing several intermediate formalisms and (automated) activities between them. Additionally,

this approach is applicable in an MPM context, as MPM naturally promotes the use of many domain-specific languages, for which we want to support the most intuitive notation, however different it is from the traditional icon-based visualizations.

Modelverse Debugging Debugging the Modelverse is complex for the same reasons why its development was difficult: its use of different interacting users and strong distinction between different services. Indeed, even a simple setup of the Modelverse requires three different programs: the MvI, MvK, and MvS. It is therefore difficult to debug all of them simultaneously: each resides (possibly) on a different machine and has its own control flow. Current debuggers are not fully up to the task of debugging the combination of several interacting programs. Additionally, even when the program can be debugged, the use of many non-deterministic intermediate components, such as the network, naturally results in non-deterministic executions. When a bug is discovered, it is therefore often difficult to exactly replicate it, let alone fix it. We proposed to not debug the generated code in a general purpose programming language, but the DEVS model instead. By using a DEVS debugger, instead of a general-purpose debugger, the level of abstraction is raised and we gain full control over the notion of time used during execution. This approach requires an MPM context due to its use of the internal models of the Modelverse. Additionally, this approach is applicable in an MPM context, as MPM tools frequently encounter problems with debugging, exactly due to its distributed and multi-user requirements.

7.1 Future Work

The work presented in this thesis can be extended and applied in many directions of future work. We present some possible future directions, some of which were already hinted on throughout this thesis.

Activity-Based Switching of Physical Implementation As the Modelverse has an explicit model of the PTM, which is used to shift most physical operations to the linguistic dimension, we mentioned that it is possible to switch the physical implementation of the Modelverse without porting the existing low-level operations. While we made two such physical implementations, one based on Python dictionaries and another one based on an RDF representation, there was no run-time changing or heterogeneous storage possible yet. Store models heterogeneously (i.e., using a combination of multiple database technologies) might be interesting, as it allows to match the PTM to the model being stored. For example, when a matrix model is stored, conforming to the matrix DSL, this can be done in a matrix-specific PTM. Other (graph-based) models can then be stored in the usual graph-based PTM. Storing these different representations, while still presenting the whole as a single graph when required, has potential to increase time and space consumption of the MvS. By linking in activity metrics, this decision can be made at runtime, and independent of interactive input from the user: the MvS itself detects access patterns and uses these to switch to the optimal internal representation dynamically.

Performance Up to now, we only considered performance assessment, and didn't consider how to improve the performance based on these results. Most of the performance

overhead is due to the MvK execution being explicitly modelled using graph transformations, requiring much communication with the MvS. While we already optimized the graph transformation engine in the generated rules, by assuming that no two rules are simultaneously applicable (as this would result in non-deterministic execution), this is not sufficient. From the assessment, it was clear that most time is spent in the MvS, due to the high number of requests made by the MvK. If we were to use a JIT compiler, thereby avoiding the need for many MvS operations for each MvK atomic operation, this number of requests can be significantly reduced. An initial JIT compiler was created by Jonathan Van der Cruysse [288], showing significant improvements already, though we believe that many further optimizations remain possible.

User Interface The user interface of a (meta-)modelling tool is mostly targeted towards usability. While we have made several contributions in the Modelverse that likely influence usability, the attached GUI is only a primitive prototype that serves as a reference implementation for the Modelverse API. Future work has to be done to make the current interfaces more user-friendly and evaluate their usability. Similarly, other existing interfaces, such as the HUTN compiler, can be developed further.

Constrained Live Modelling Sanitization We have presented a structured approach to live modelling, where all live modelling related operations were lifted to a single sanitization activity. Language engineers then only have to additionally implement a sanitization activity to make their domain-specific language live. This activity, however, is currently unconstrained apart from the meta-model signature (i.e., input and output models must conform to the meta-model). Indeed, now it is possible to implement arbitrary behaviour in this sanitization activity, not necessarily related to live modelling (e.g., set the simulation time to a different value). In future work, this activity could be further constrained to actually guarantee that it makes sense, for example by forcing that simulation time values must remain untouched.

Interaction Models for the GUI In our presentation of a more flexible approach to concrete syntax, we have explicitly moved all perceptualization logic to the back-end, only keeping the rendering logic in the front-end. And while we have also defined a recognition and comprehension phase, taking in changes from the front-end and propagating them to the back-end, we have not mentioned how the front-end manipulates the model. For example, which sequence of input events is required to drag around an element on the canvas? Current tools restrict themselves to a generic interface: left-click, move the mouse, and then left-release are what is required to move an element. This might not be intuitive for all domain-specific languages, though: if the DSL has no notion of moving elements, the front-end might have to disallow moving elements altogether. Similarly, some domain-specific languages might have another way to change the location of an element, which is more intuitive in that domain. In this case, the interaction is part of the domain-specific language, and should be created by the language engineer. In summary, the language engineer should not only have full control over the concrete syntax, but also over the interaction with said concrete syntax. In future work, we project that the language engineer creates not only a concrete syntax model, but includes an interaction model, which is used by the front-end.

Appendix A

Modelverse State Specification

This chapter introduces a formalized description of the specification of the Modelverse State (MvS). Assuming that an informal description is already known, as presented in this thesis, the specification of the various operations is given.

A.1 Data representation

The Modelverse State defines a graph $G = \langle N_G, E_G, N_{V,G} \rangle$, element of $\mathcal{G}$ (the set of all possible states of the MvS). It consists of nodes (identifiers stored in N_G), possibly with values $\in \mathbb{U}$ defined on them through the mapping $N_{V,G} : N_G \rightarrow \mathbb{U}$, and edges, stored as triples $\{\langle s, id, t \rangle\} \in E_G$. We additionally define the set of edge identifiers as $E_{IDS,G} = \{id \| \langle s, id, t \rangle \in E_G\}$. An edge is identified by its identifier, such that $\forall e_i, e_j \in E : e_i = (a, b, c), e_j = (d, e, f), (b = e) \Rightarrow (a = d) \wedge (c = f)$. The combined set of all identifiers is termed $IDS_G = N_G \cup E_{IDS,G}$ and there is no overlap between both sets ($N_G \cap E_{IDS,G} = \emptyset$). Nodes and edges have a unique identifier, with IDS_G being (exactly) the set of all identifiers.

$\mathbb{U}$ defines the set of all possible node values and is the union of all possible types: $\mathbb{U} = \mathbb{I} \cup \mathbb{F} \cup \mathbb{S} \cup \mathbb{B} \cup \mathbb{A} \cup \Sigma_{type}$. We define the following primitive types, supported in the PTM, for which the MvS provides native support:

- **Integer** ($\mathbb{I}$) as the set of integers in the range $[-2^{63}, 2^{63} - 1]$;
- **Float** ($\mathbb{F}$) as the set of double precision floating point numbers;
- **String** ($\mathbb{S}$) as the set of all ordered combinations of ASCII characters;
- **Boolean** ($\mathbb{B}$) as either True or False;
- **Action** ($\mathbb{A}$) as an action language construct, used to define Modelverse semantics. These are { if, while, assign, call, break, continue, return, resolve, access, constant, declare, global, input, output }.

We use $\mathbb{I}$ and $\mathbb{F}$, instead of $\mathbb{Z}$ and $\mathbb{R}$, respectively, for practical reasons, as this is closer to the implementation level and allows for more efficient implementations. By enforce the size of

data values, we prevent implementation-dependent behaviour (*e.g.*, some implementation using 32-bit integers, whereas another uses 64-bit integers).

A subgraph $M = \langle N_M, E_M, N_{V,M}\rangle$ of a graph $G = \langle N_G, E_G, N_{V,G}\rangle$, denoted as $M \subseteq G$ as a graph containing a subset of the nodes ($N_M \subseteq N_G$) and edges ($E_G \subseteq E_G$), where all used nodes node values are copied as well ($\forall (a, b, c) \in E_M : a \in IDS_M \wedge c \in IDS_M$ and $\forall (a \to b) \in N_{V,G} : a \in N_M \Rightarrow (a \to b) \in N_{V,M}$). It is implicit that the resulting graph should still be valid according to the restrictions placed on the graph (*e.g.*, source and target of nodes is still present).

A.2 CRUD interface

The interface provides the various operations that are available. Each reply is furthermore annotated with a status code (S), indicating whether the operation was successful ($s = 100$), if syntactical preconditions were invalid ($s = 2xx$), or if semantical preconditions were invalid ($s = 3xx$).

A.2.1 Create

First is the create node operation (C_N), which takes no arguments and returns the identifier of the newly created node, which was unused up to now.

$$C_N : \mathcal{G} \to \mathcal{G} \times N \times S$$
$$C_N(\langle N, E, N_V\rangle) = (\langle N \cup \{n\}, E, N_V\rangle, n, 100)$$
$$n \notin IDS$$

The create edge operation (C_E) takes the identifier of the source and target elements (either a node or an edge) as argument, and returns the identifier of the newly created edge.

$$C_E : \mathcal{G} \times IDS \times IDS \to \mathcal{G} \times E_{IDS} \times S$$
$$C_E(\langle N, E, N_V\rangle, i_1, i_2) =$$
$$\begin{cases} (\langle N, E \cup \{(i_1, i_3 \notin IDS, i_2)\}, N_V\rangle, i_3, 100) & \textit{if} \quad i_1 \in IDS \wedge i_2 \in IDS \\ (\langle N, E, N_V\rangle, \textit{None}, 200) & \textit{if} \quad i_1 \notin IDS \\ (\langle N, E, N_V\rangle, \textit{None}, 201) & \textit{if} \quad i_2 \notin IDS \end{cases}$$

The last primitive create operation (C_{NV}) creates a new node, and assigns it a value immediately. It has the same signature as the create node, but takes a primitive value to assign to the created node. This operation could be implemented by first creating an empty node and afterwards updating its value, though this would negatively impact performance.

$$C_{NV} : \mathcal{G} \times \mathbb{U} \to \mathcal{G} \times N \times S$$
$$C_{NV}(\langle N, E, N_V \rangle, d) = \begin{cases} (\langle N \cup \{i\}, E, N_V \cup (i \to d) \rangle, i, 100) & \textit{if} \quad d \in \mathbb{U} \\ (\langle N, E, N_V \rangle, \textit{None}, 202) & \textit{if} \quad d \notin \mathbb{U} \end{cases}$$
$$i \notin N$$

For performance, we add a composite create operation, which creates a named edge between two graph elements (C_D). This operation is equivalent to creating an edge between the two elements, followed by creating an edge from the newly created edge, to the data value that was specified. It is formalised as follows.

$$C_D : \mathcal{G} \times IDS \times \mathbb{U} \times IDS \to \mathcal{G} \times S$$
$$C_D(\langle N, E, N_V \rangle, a, d, b) =$$
$$\begin{cases} (\langle N, E, N_V \rangle, 203) & \textit{if} \quad a \notin IDS \\ (\langle N, E, N_V \rangle, 204) & \textit{if} \quad d \notin \mathbb{U} \\ (\langle N, E, N_V \rangle, 205) & \textit{if} \quad b \notin IDS \\ (\langle N \cup \{c\}, E \cup \{(a, i_1, b), (i_1, i_2, c)\}, N_V \cup \{(c \to d)\} \rangle, 100) & \textit{else} \end{cases}$$
$$c, i_1, i_2 \notin IDS$$

A.2.2 Read

The next set of operations consists of read operations. As there is no useful information in non-data nodes, there is no read operation defined on nodes, except for their primitive data (R_V). It is an error if the node that is being read does not have a value assigned to it.

$$R_V : \mathcal{G} \times N \to \mathbb{U} \times S$$
$$R_V(\langle N, E, N_V \rangle, n) = \begin{cases} (\textit{None}, 206) & \textit{if} \quad n \notin N \\ (\textit{None}, 300) & \textit{if} \quad n \notin dom(N_V) \\ (N_V(n), 100) & \textit{else} \end{cases}$$

Instead of a read operation on the nodes, it is possible to read out their outgoing edges (R_O) and incoming edges (R_I). This works for nodes, but also for edges, as edges can also be the source (and target) of other edges. The result is the identifier of the connected edges, in an unordered collection.

$$R_O : \mathcal{G} \times IDS \to 2^E \times S$$
$$R_O(\langle N, E, N_V \rangle, i) = \begin{cases} (\{(i, b, c) \in E, 100) & \textit{if} \quad i \in IDS \\ (\textit{None}, 207) & \textit{if} \quad i \notin IDS \end{cases}$$

$$R_I : \mathcal{G} \times IDS \to 2^E \times S$$
$$R_I(\langle N, E, N_V \rangle, i) = \begin{cases} (\{(a, b, i) \in E, 100) & \textit{if} \quad i \in IDS \\ (\textit{None}, 207) & \textit{if} \quad i \notin IDS \end{cases}$$

A read operation for edges (R_E) is defined as returning a tuple consisting of the source and target of the edge. Due to the restriction on the edge identifier, both the source and target identifier will be smaller than the edge identifier.

$$R_E : \mathcal{G} \times E_{IDS} \to IDS \times IDS \times S$$
$$R_E(\langle N, E, N_V \rangle, i_1) = \begin{cases} (\textit{None}, \textit{None}, 209) & \textit{if} \quad i_1 \notin E_{IDS} \\ (i_2, i_3, 100) & \textit{else} \end{cases}$$
$$e = (i_2, i_1, i_3) \in E$$

For efficiency, an additional *"dictionary read"* operation (R_{dict}) is defined to read an element which is linked to another one through an edge, which is connected to a node with a primitive value. This allows for a more efficient implementation of lookups from a specific node, without requiring an exhaustive search of the connected edges. While the search might still be necessary internally, implementations are free to create specialized data structures for this operation. Even if that is not the case, this operation reduces the amount of calls required to 1. If the specified entry is not found in the dictionary, an error is raised.

Notice that there is room for ambiguity if a node has multiple outgoing links, linking to the same data value. While this could cause an error, we explicitly allow for this situation for performance reasons, as otherwise the search would always need to traverse all links, even if a match was already found. Similarly, multiple outgoing edges might exist with the same label added to them, resulting in ambiguity. For performance reasons, however, the result will be non-deterministic.

$$R_{dict} : \mathcal{G} \times IDS \times \mathbb{U} \to IDS \times S$$
$$R_{dict}(\langle N, E, N_V \rangle, i_1, v) =$$
$$\begin{cases} (\textit{None}, 210) & \textit{if} \quad i_1 \notin IDS \\ (\textit{None}, 211) & \textit{if} \quad v \notin \mathbb{U} \\ (\textit{None}, 301) & \textit{if} \quad \nexists b, c \in E_{IDS} : (i_1, b, i_2), (b, c, N_V(v)) \in E \\ (i_2, 100) & \textit{else} \end{cases}$$

Some other, more complex read operations on dictionaries are also supported, purely for efficiency reasons. Their errors are similar to the R_{dict} operation. Each of these operations returns a slightly different result, determined by the frequently used operations in the next section. These operations are:

- R_{dict_node} returns the element being linked to, but instead of a primitive value, it searches for a specific element by identifier. It therefore does not try to dereference the value stored in the resulting element, nor will it match different elements with the same value.
- R_{dict_edge} is equivalent as R_{dict}, but returns the identifier of the edge between them, instead of the element itself.
- $R_{dict\_reverse}$ returns a list of all elements that have an outgoing link towards the passed element, with the provided name on that edge. It is therefore basically a reverse dictionary lookup: return the dictionaries that contain this exact value with a specified key.

Multiple combinations would also be possible, though we only formalize those that are used by the MvK.

$$R_{dict\_keys} : \mathcal{G} \times IDS \to 2^{IDS} \times S$$
$$R_{dict\_keys}(\langle N, E, N_V \rangle, a) =$$
$$\begin{cases} (\textit{None}, 222) & \textit{if} \quad i_1 \notin IDS \\ (\{e | (a, b, c), (b, d, e) \in E\}, 100) & \textit{else} \end{cases}$$

$$R_{dict\_node} : \mathcal{G} \times IDS \times IDS \to IDS \times S$$
$$R_{dict\_node}(\langle N, E, N_V \rangle, i_1, d) =$$
$$\begin{cases} (\textit{None}, 212) & \textit{if} \quad i_1 \notin IDS \\ (\textit{None}, 213) & \textit{if} \quad i_2 \notin IDS \\ (\textit{None}, 303) & \textit{if} \quad \nexists b, c \in E_{IDS} : (i_1, b, i_2), (b, c, d) \in E \\ (i_2, 100) & \textit{else} \end{cases}$$

$$R_{dict\_edge} : \mathcal{G} \times IDS \times \mathbb{U} \to IDS \times S$$
$$R_{dict\_edge}(\langle N, E, N_V \rangle, i_1, v) =$$
$$\begin{cases} (\textit{None}, 214) & \textit{if} \quad i_1 \notin IDS \\ (\textit{None}, 215) & \textit{if} \quad v \notin \mathbb{U} \\ (\textit{None}, 305) & \textit{if} \quad \nexists b, c \in E_{IDS} : (i_1, i_2, b), (i_2, c, N_V(v)) \in E \\ (i_2, 100) & \textit{else} \end{cases}$$

$$R_{dict\_node\_edge} : \mathcal{G} \times IDS \times IDS \to IDS \times S$$
$$R_{dict\_node\_edge}(\langle N, E, N_V \rangle, i_1, d) =$$
$$\begin{cases} (\textit{None}, 216) & \textit{if} \quad i_1 \notin IDS \\ (\textit{None}, 217) & \textit{if} \quad i_2 \notin IDS \\ (\textit{None}, 307) & \textit{if} \quad \nexists b, c \in E_{IDS} : (i_1, i_2, b), (i_2, c, d) \in E \\ (i_2, 100) & \textit{else} \end{cases}$$

$$R_{dict\_reverse} : \mathcal{G} \times IDS \times \mathbb{U} \to 2^{IDS} \times S$$
$$R_{dict\_reverse}(\langle N, E, N_V \rangle, i_1, v) =$$
$$\begin{cases} (\textit{None}, 218) & \textit{if} \quad i_1 \notin IDS \\ (\textit{None}, 219) & \textit{if} \quad v \notin \mathbb{U} \\ (\textit{None}, 309) & \textit{if} \quad \nexists b, c \in E_{IDS} : (i_1, b, i_2), (b, c, d) \in E \\ (\{i_2 : (i_2, b, i_1), (b, c, d) \in E\}, 100) & \textit{else} \end{cases}$$

A.2.3 Update

Even though we implement a CRUD interface, we do not offer support for any update operations.

The most important reason is correctness and performance. Updating the source and target of edges has the potential of creating impossible loops, like an edge connecting itself. While this is impossible to do when constructing the edge at first (as it is required that its source and target already exist), this can no longer be guaranteed when the edge is updated. An alternative would be to allow updates, but search for correctness violations (*i.e.*, recursively following the source and target of an edge, we ultimately end up in nodes) after the update was done. This would have a significant, and unpredictable, impact on performance when performing an update for an edge. As an update operation is similar to a subsequent create and delete, which have better complexity, we did not think this is a viable approach. Yet another alternative would be to allow updates again, but only those updates that change the source and target to nodes that existed when the edge was originally created. This prevents correctness violations by construction, though it does not make the operation generally applicable. And since we would need to have a fallback method (*i.e.*, subsequent delete and create) anyway, it might be easier to just always use the fallback method. This also prevents us having to store some kind of causality information, like which elements were created before which other elements.

Another reason is cache management, as also proposed by [192]. If a node can be updated, caches can become invalid, implying some kind of MvS-initiated invalidation protocol for the MvK. While we do not have any significant optimization for this yet, restricting updates has significant potential.

A.2.4 Delete

Finally there are the *delete* operations. The source and target of each edge should always exist in the graph. Therefore, if a deleted node or edge is the source or target of an edge, the edge needs to be recursively removed. The resulting graph should thus be the largest possible subgraph of the original graph, while still being a valid graph. For the delete node operation (D_N), the node itself is removed, and then all connected edges are recursively removed.

$$D_N : \mathcal{G} \times N \to \mathcal{G} \times S$$

$$D_N(G = \langle N, E, N_V \rangle, i) = \begin{cases} (G, 220) & \textit{if} \quad i \notin N \\ (\langle N \setminus \{i\}, E', N'_V \rangle = G' \subset G, 100) & \textit{else} \end{cases}$$

$$\forall G'' \subseteq G : (G' \subseteq G'') \Rightarrow G' = G''$$

The delete edge operation (D_E) operation is similar, but it is guaranteed that no nodes are removed at all. Due to recursive deletions, the resulting set of edges is possibly a subset of the original edges. The resulting graph is again the largest possible (valid) subgraph, with the specified edge removed.

$$D_E : \mathcal{G} \times E_{IDS} \to \mathcal{G} \times S$$

$$D_E(G = \langle N, E, N_V \rangle, i) =$$

$$\begin{cases} (G, 221) & \textit{if} \quad i \notin E_{IDS} \\ (\langle N, E' \subseteq E \setminus \{(a, i, c) \in E, N'_V \rangle = G' \subset G, 100) & \textit{else} \end{cases}$$

$$\forall G'' \subseteq G : (G' \subseteq G'') \Rightarrow G' = G''$$

Appendix B

Action Language Specification

The complete Action Language specification is given below. The Action Language specification is based on graph transformation rules. These rules are modelled explicitly in the Modelverse, and are used to automatically generate Graphviz files for documentation (Section B.1), and synthesize a Python implementation.

B.1 Documentation

We now introduce all graph transformation rules that are used by the Modelverse Kernel to execute the action code. For each of the figures, we assume that the Modelverse root node has only a unique match, and is shown in all rules as the topmost node. As we will see, there is at most one rule applicable, meaning that there is no non-determinism. Additionally, there will always be an applicable rule, given that the action language code and functions are well-defined (e.g., the required outgoing links are defined, all function definitions have a body attached). For each of the rules, we briefly provide some pointers as to what they mean.

B.1.1 If condition

The *If* construct will first evaluate the condition (*cond* link) by moving the instruction pointer there. It signals that it should be executed again afterwards, but now in phase *cond*, by putting this on the evaluation stack (Figure B.1). As soon as the condition is evaluated, and the *If* popped back from the stack, the return value (of the condition) can either be True or False. If it is True (Figure B.2), the *then* link is executed, and the *if* is pushed on the stack again, but now in the final phase *finish*. This is the phase which signals to another rule that this operation has finished, and the next instruction can be loaded. If it is False, and there is an *else* link (Figure B.3), it is executed, similar to the previous case. If it is False, but there is no *else* link (Figure B.4), the *If* is marked as completed immediately, without any subsequent actions.

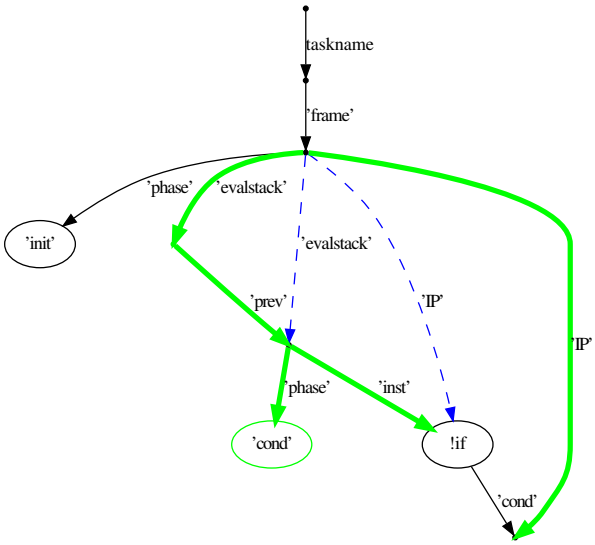


Figure B.1: *If* construct needs to evaluate the condition.

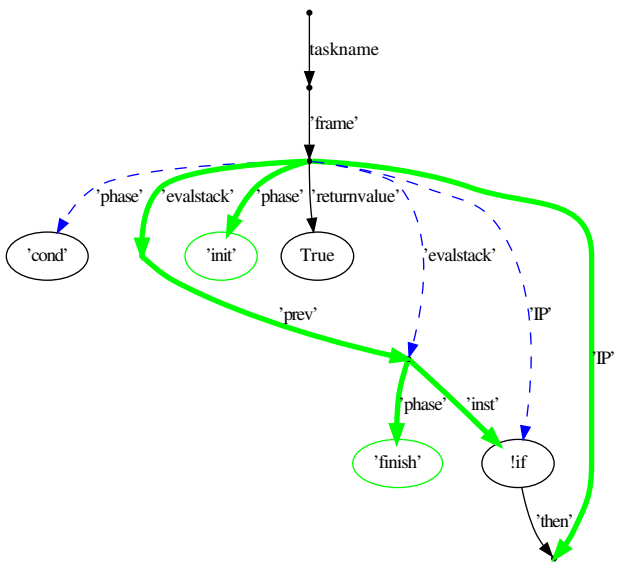


Figure B.2: *If* construct needs to evaluate the then branch.

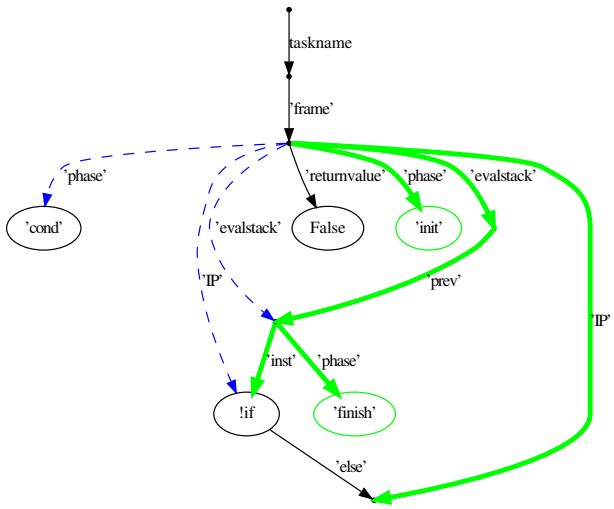


Figure B.3: *If* construct needs to evaluate the else branch, and there is one.

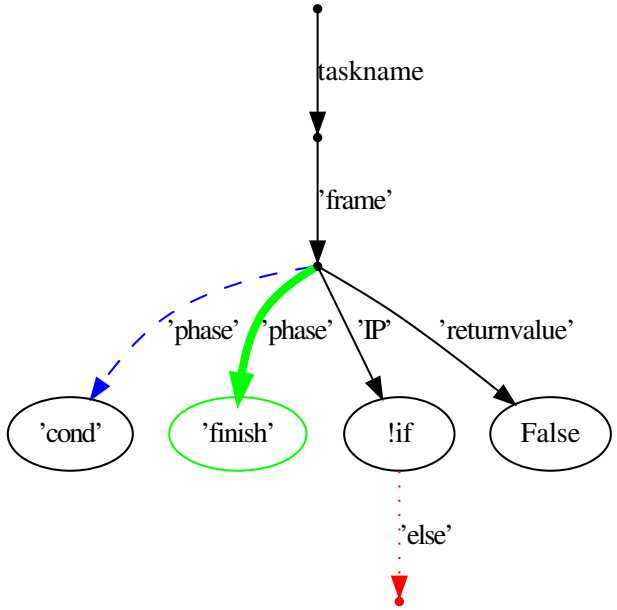


Figure B.4: *If* construct needs to evaluate the else branch, but there is none.

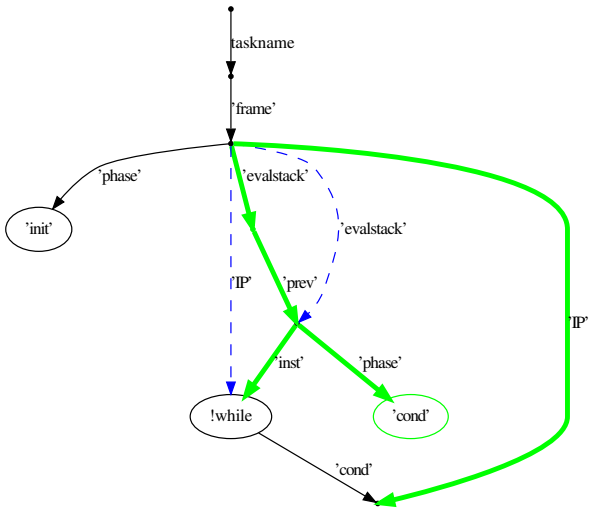


Figure B.5: *While* construct needs to evaluate the condition.

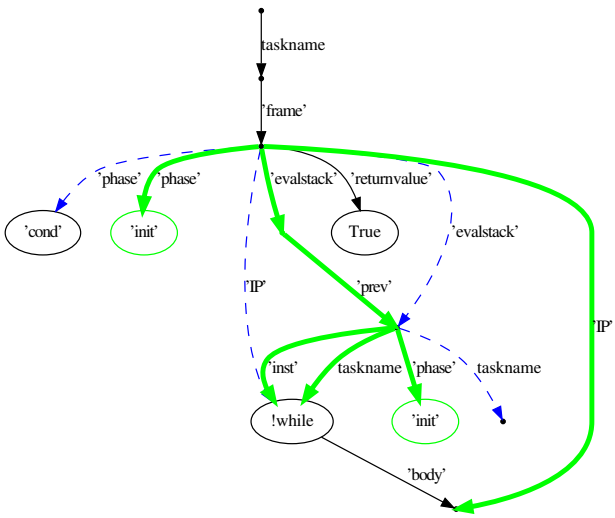


Figure B.6: *While* construct must loop.

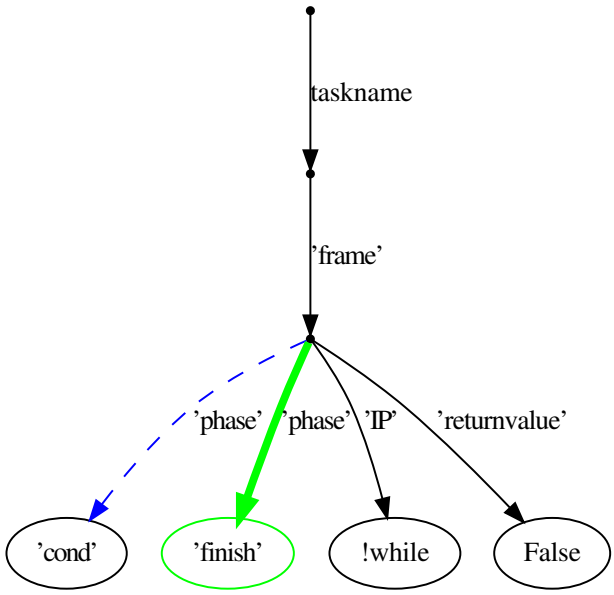


Figure B.7: *While* construct must terminate.

B.1.2 While loop

The *While* construct will first evaluate the condition (*cond* link) by moving the instruction pointer there. It signals that it should be executed again afterwards, but now in phase *cond*, by putting this on the stack (Figure B.5). As soon as the condition is evaluated, and the *While* popped from the stack, the return value (of the condition) can either be True or False. If it is True (Figure B.6), the *body* link is executed, and the *While* is pushed on the stack again, but with its phase set to *init*. This way, the while construct will again be executed after the body has terminated. By setting the phase to *init*, we effectively cause looping, as the condition will again be evaluated, and, depending on the result, the body gets executed once more. If it is False (Figure B.7), the *While* is immediately marked as finished and the body is not executed.

B.1.3 Break

The *Break* construct will move the instruction pointer back to the *While* construct it belongs to (Figure B.8). The phase is set to *finish* to indicate that the loop has finished. This prevents the condition evaluation and marks the end of the while loop.

B.1.4 Continue

The *Continue* construct will move the instruction pointer back to the *While* construct to which it belongs (Figure B.9). The phase is set to *init* to indicate that the loop needs to continue. This causes the condition to be evaluated again, indicating the next iteration of the loop.

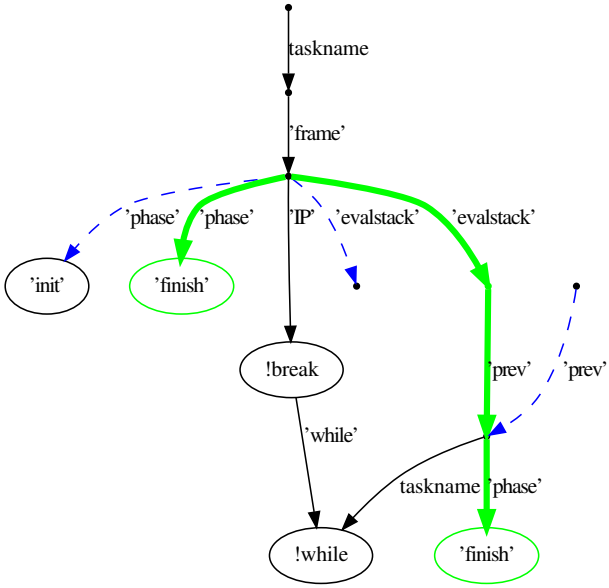


Figure B.8: *Break* construct.

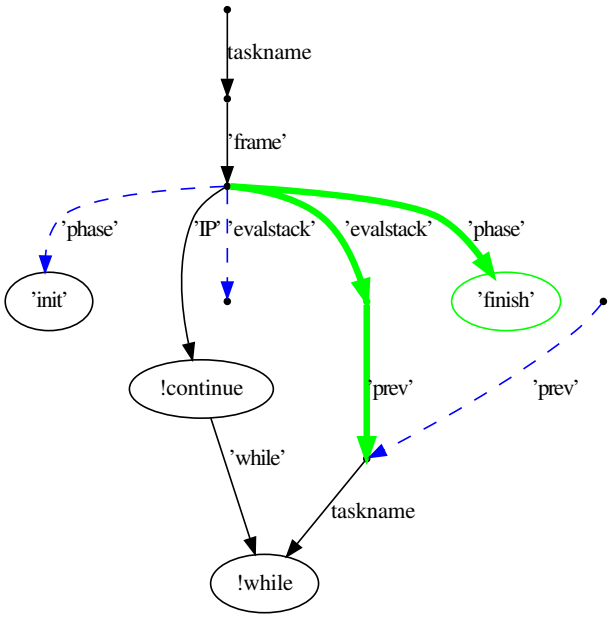


Figure B.9: *Continue* construct.

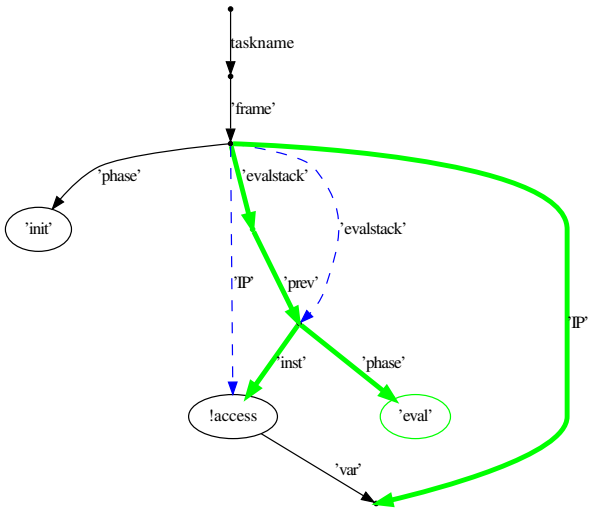


Figure B.10: *Access* construct must fetch the referred value.

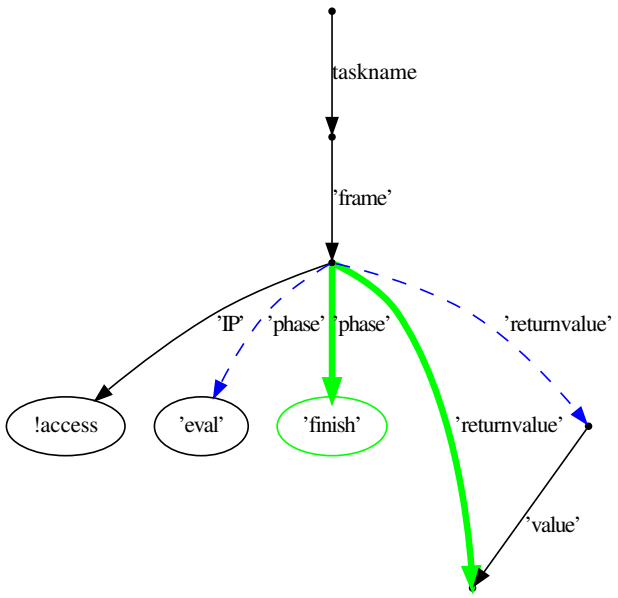


Figure B.11: *Access* construct needs to access the value.

B.1.5 Access

The *Access* construct will move the instruction pointer to the variable which has to be resolved first (Figure B.10). It signals that it needs to be executed again after the variable was resolved, by putting itself on the evaluation stack. After resolution of the variable, the value of the variable is accessed and set as the new return value (Figure B.11).

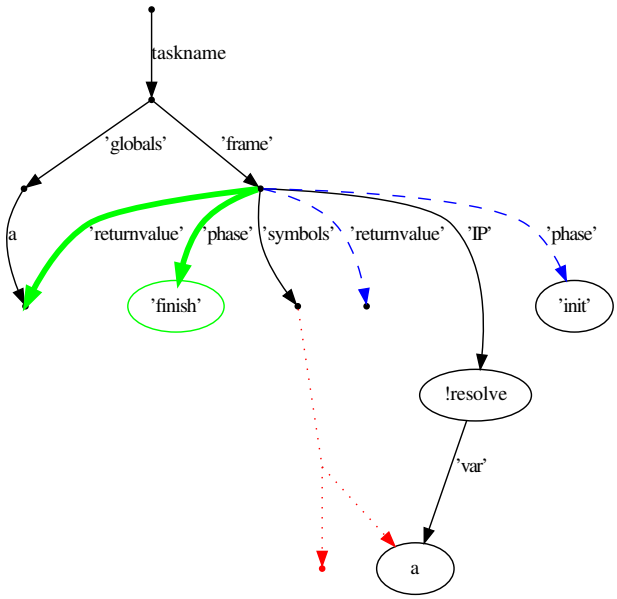


Figure B.12: *Resolve* construct resolves a non-global element.

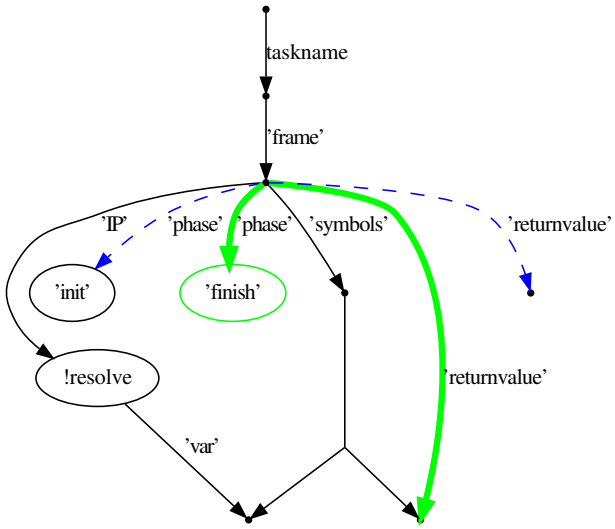


Figure B.13: *Resolve* construct resolves a global element.

B.1.6 Resolve

With the *resolve* rule, a variable is looked up in either the local (Figure B.12) or global (Figure B.13) symbol table. The variable in the symbol table will be set as the returnvalue. The local symbol table has priority over the global symbol table. Note that the returned value is only a reference, similar to the lvalue in parsers. A further *Access* is required to read out the actual value.

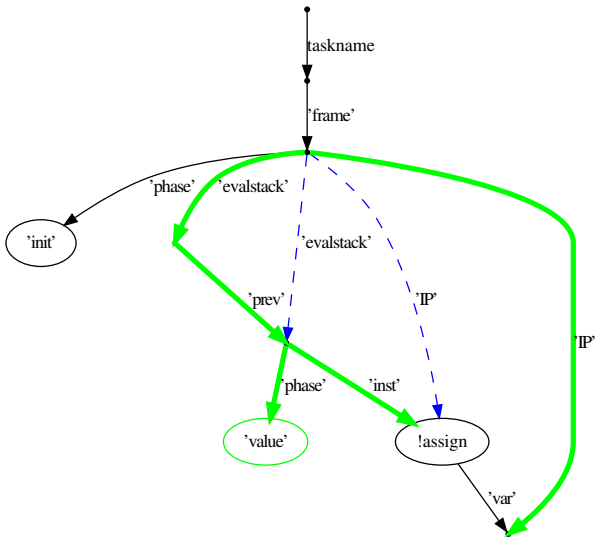


Figure B.14: *Assign* construct reads out the symbol to assign to.

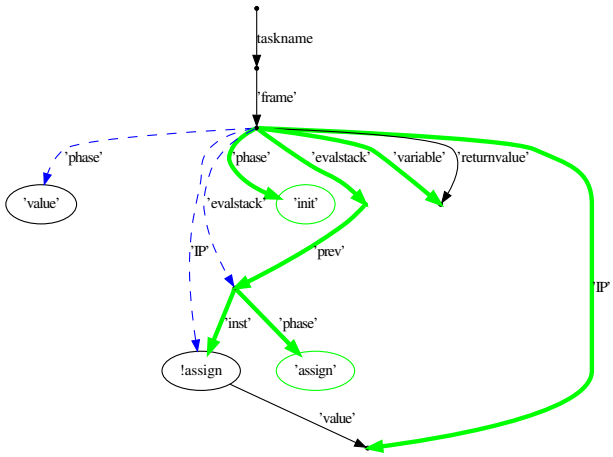


Figure B.15: *Assign* construct reads out the value to assign.

B.1.7 Assign

The *Assign* rule will first evaluate the variable (Figure B.14), as it will first need to be resolved. After resolution (Figure B.15), the found value is stored in a temporary link from the frame (*variable* link). The instruction pointer is moved to the value that will be assigned, as it will also need to be evaluated. After the value is evaluated (Figure B.16), the value link in the stored variable is changed to the evaluated value.

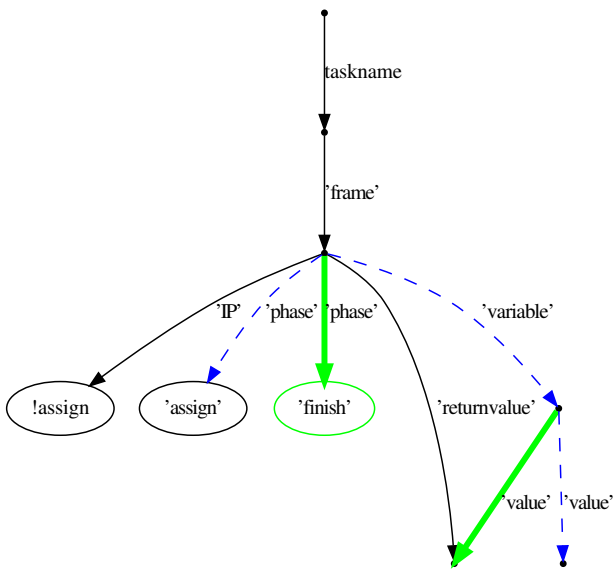


Figure B.16: *Assign* construct performs the actual assigment.

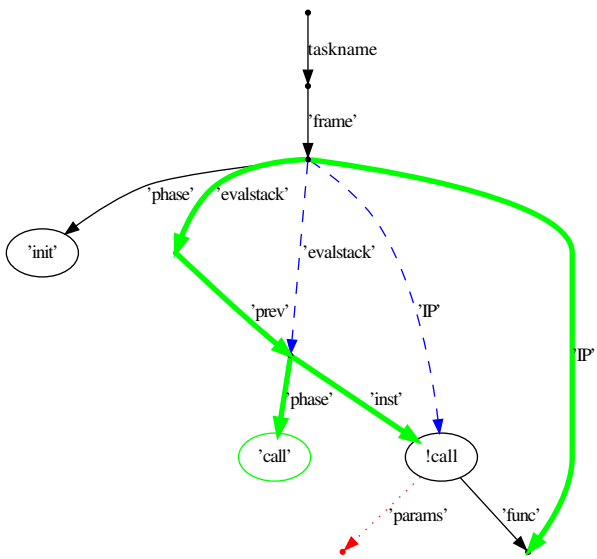


Figure B.17: *Call* construct resolves function with no parameters.

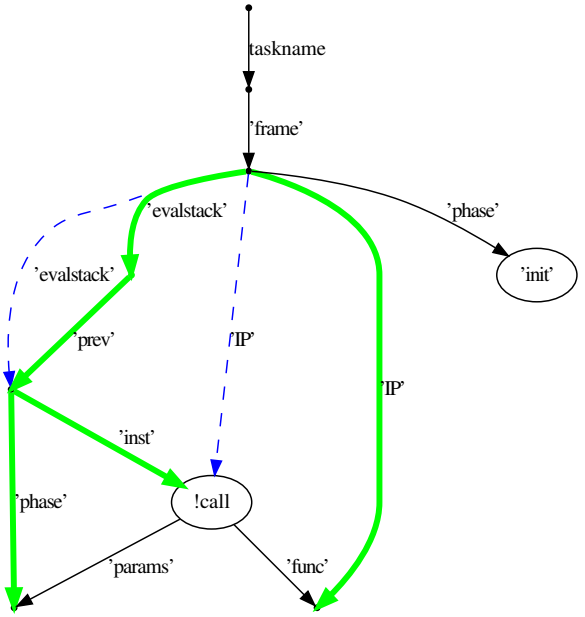


Figure B.18: *Call* construct resolves function with parameters.

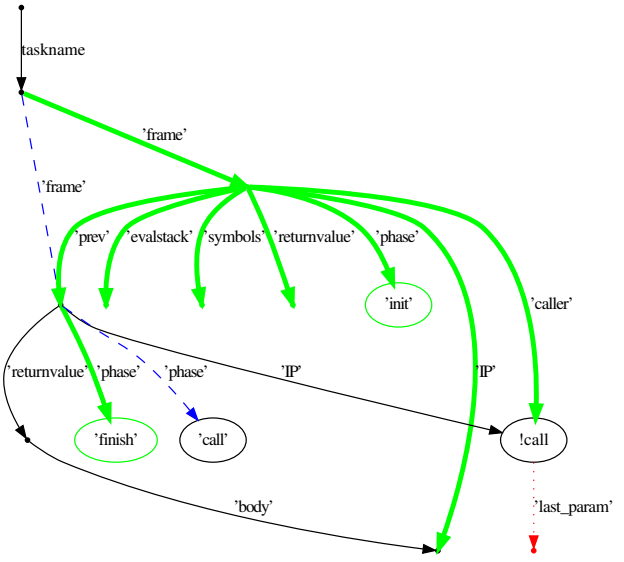


Figure B.19: *Call* construct invokes with no parameters.

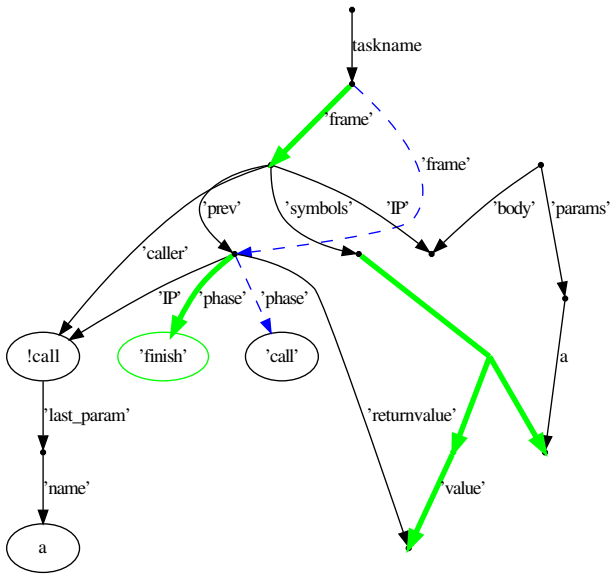


Figure B.20: *Call* construct invokes with parameters.

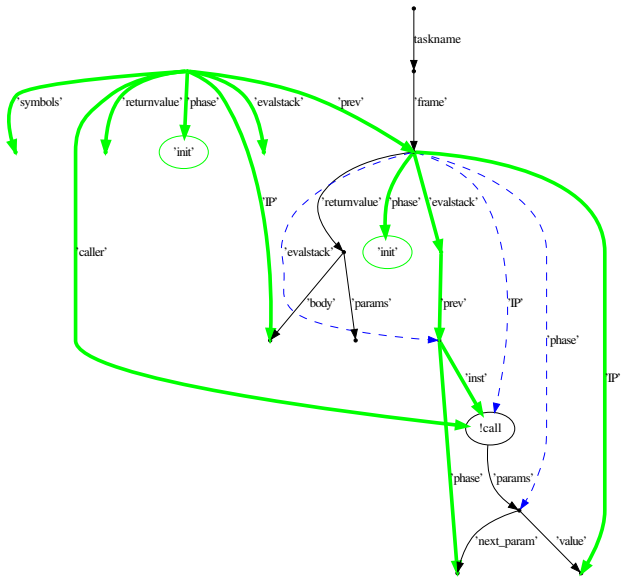


Figure B.21: *Call* construct evaluates first of multiple parameters.

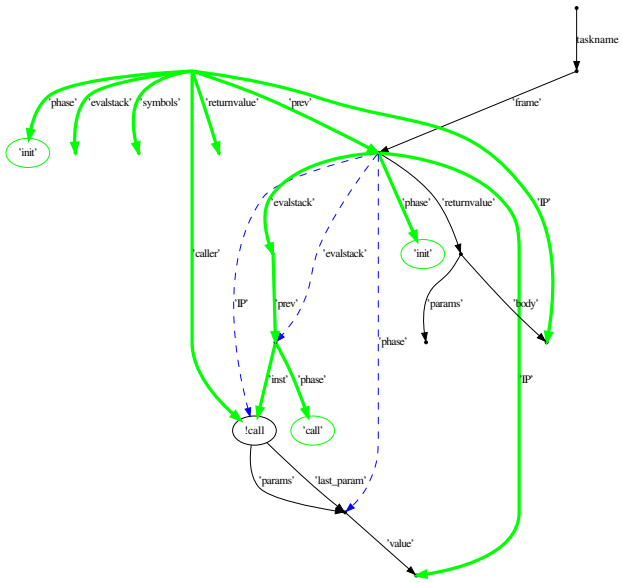


Figure B.22: *Call* construct evaluates first and only parameter.

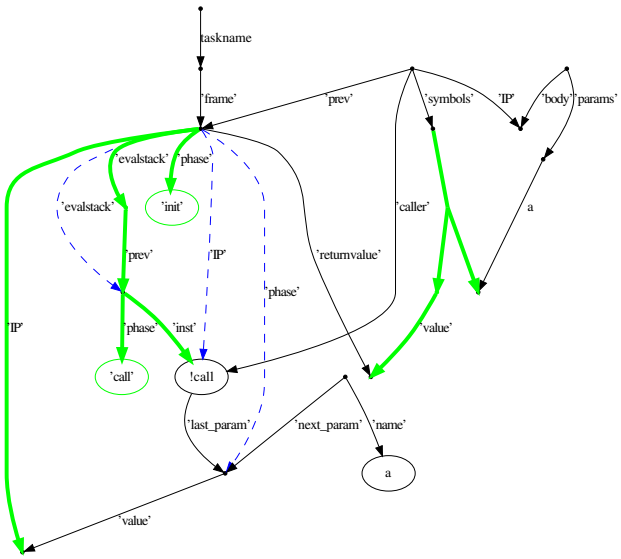


Figure B.23: *Call* construct evaluates last of multiple parameters.

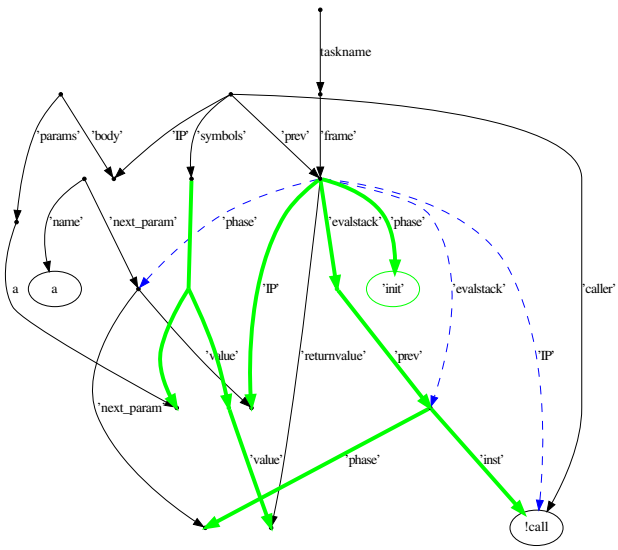


Figure B.24: *Call* construct evaluates next of multiple parameters.

B.1.8 Function Call

A *Call* construct has different paths, depending on how many parameters there are. The distinct situations are:

1. **No parameters**: in this simple case, the method is first resolved by moving the instruction pointer there, and the call is already put on the stack (Figure B.17). After the function is resolved (Figure B.19), the call is made by creating a new execution frame and making it the active frame.

2. **One parameter**: similar to the previous situation, the function is first resolved (Figure B.18), but instead of putting the *call* on the stack, the first parameter is used. Afterwards (Figure B.22), the stack is created for the resolved function, the instruction pointer is set to evaluate the argument, and the *call* is put on the stack. When the parameter is evaluated (Figure B.20), the result is put in the symbol table of the new execution frame and the new frame is made active.

3. **Two parameters**: similar to a single parameter, the first parameter is again put on the stack for after the function resolution (Figure B.19). When evaluating the first parameter (Figure B.21), the *next_param* parameter is put on the stack, instead of the *call* phase. The second parameter is already the last parameter, so we then put the *call* on the stack (Figure B.23). Finally, the function is called as with only a single parameter (Figure B.20).

4. **More than two parameters**: similar to two parameters, but with an iteration rule (Figure B.24) for all parameters except the first and last. This iteration rule simply evaluates the parameters in order of their *next_param* links.

In all cases, the *finish* is put on the stack during the call to the function. As soon as the called function has finished, it will invoke a return and thus pop the active execution frame.

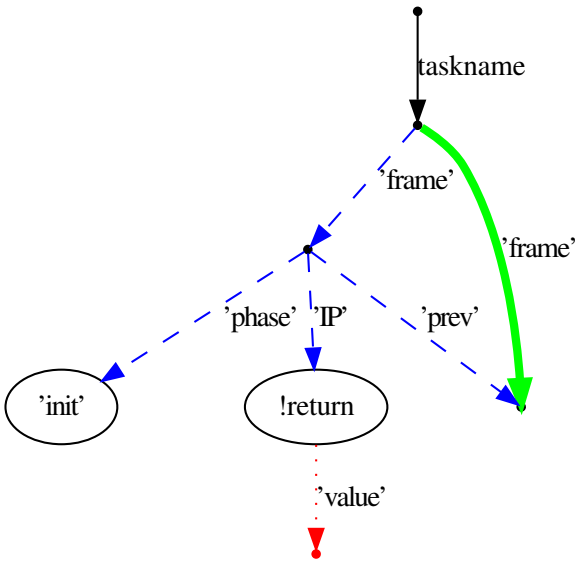


Figure B.25: *Return* construct returns with no returnvalue.

This will make the current frame active again, which will then progress towards the next instruction.

Parameter passing happens through the use of both named variables and positional parameters. However, the positional parameters are only used to determine the evaluation order, and not for binding of actual to formal parameter. It is possible for a front-end to offer positional parameters, by automatically mapping them onto their formal parameters.

B.1.9 Return

For the *Return* construct, there are again two options: either there is a value to return, or there is none. If there is no return value (Figure B.25), the current execution frame is removed and the previous one is made active again, without touching the return value of the underlying frame. If there is a return value (Figure B.26), it is first evaluated by moving the instruction pointer there. After evaluation (Figure B.27), the evaluated value is stored in the returnvalue of the previous frame, and the current frame is deleted.

B.1.10 Constant

The *Const* construct is used for constants, which are closely linked to the primitive data types presented in the Modelverse State. It is only used as an 'executable wrapper' for a literal: evaluation of this construct will yield the contained node (Figure B.28). The phase is also set to *finish*, to indicate termination of the construct.

B.1.11 Declare

The Declare instruction (Figure B.29) will add the specified node to the symbol table, so that it can be assigned a value, or read out. As the declare does not take a value, the default

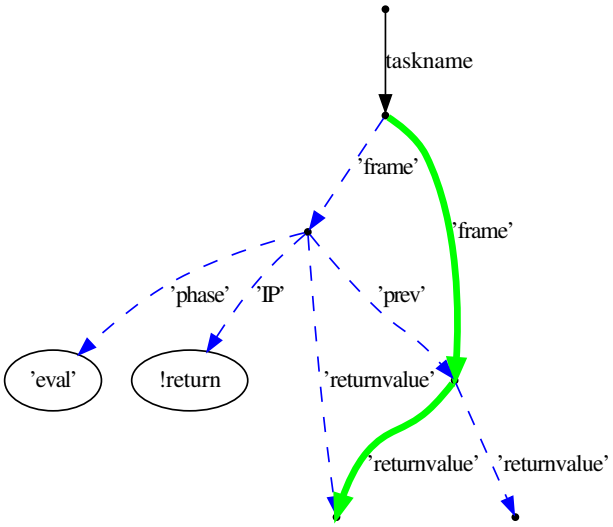


Figure B.26: *Return* construct evaluates the returnvalue.

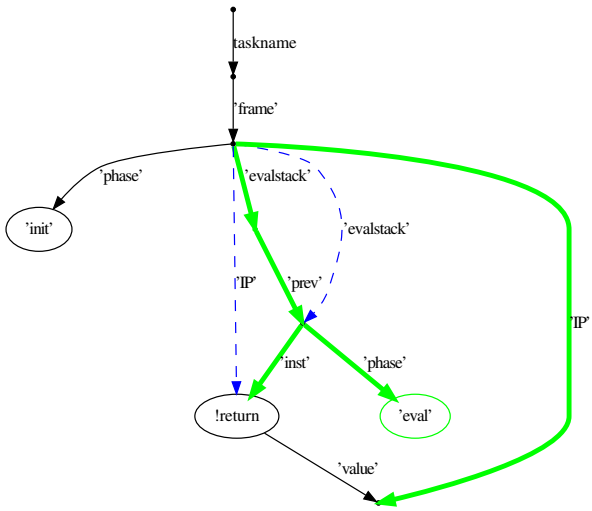


Figure B.27: *Return* construct returns the evaluated returnvalue.

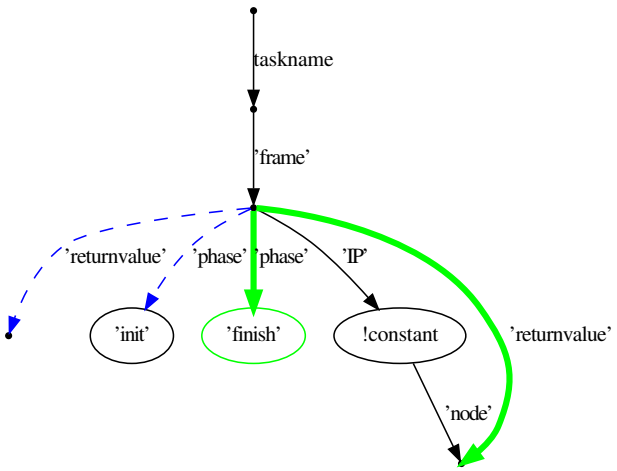


Figure B.28: *Constant* construct.

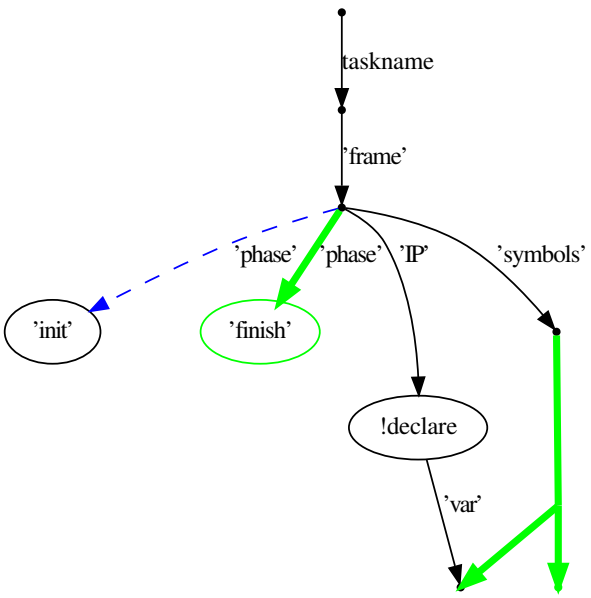


Figure B.29: *Declare* construct for a local variable.

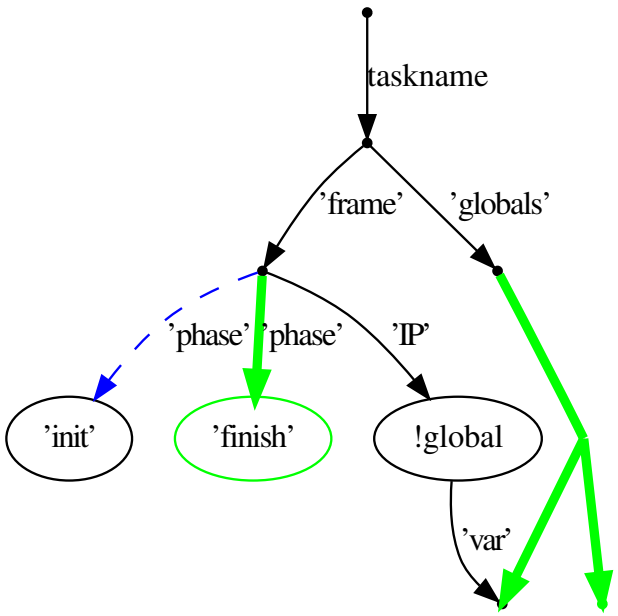


Figure B.30: *Global* construct for a global variable.

value of the node is just an empty node. Future instructions can use the node connected to the Declare instruction to reference to the variable.

Apart from a declaration in the symbol table of the current user, it is also possible to declare it in the global namespace (Figure B.30). This makes sure that other users can also find it and access the values. Its primary use will be function resolution though, as functions should be declared in a higher scope than the current scope. Nonetheless, it is possible to define everything else as a global too, making it accessible.

B.1.12 I/O

The *Output* construct will first evaluate the element the 'value' link points to (Figure B.32), and afterwards it puts the returnvalue in the output queue (Figure B.33).

The *Input* construct will read the value that is in the input queue and put it in place of the returnvalue. No evaluation whatsoever is done on the values (Figure B.31).

B.1.13 Control Instructions

When the instruction pointer points to an instruction which is marked as *finish*ed, one of these helper rules becomes active. These are responsible for progressing towards the next instruction. Either there is a *next* link (Figure B.34), which links towards the next instruction to execute. If it is present, the instruction pointer is moved to this instruction, and the phase is reset to *init* as it is the first time this construct is executed. In case no *next* link exists (Figure B.35), the next instruction is popped from the stack, together with its

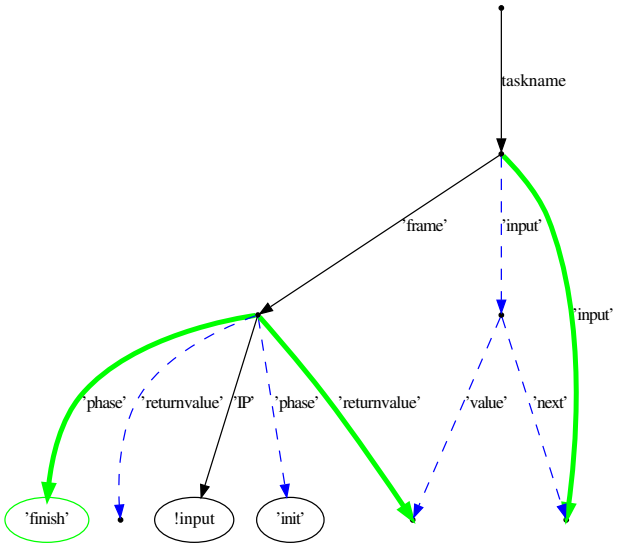


Figure B.31: *Input* construct for fetching external input.

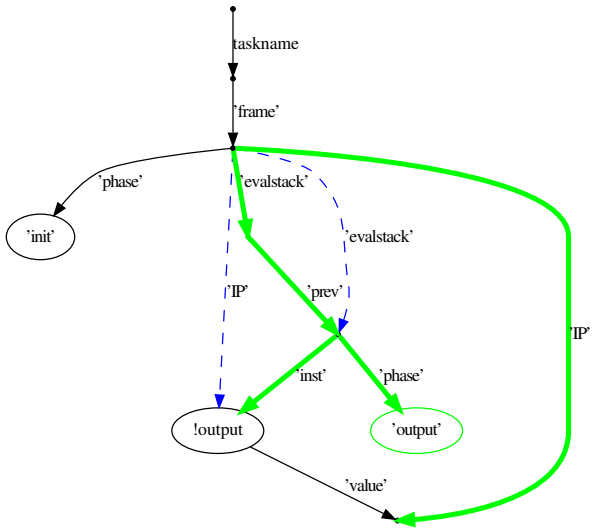


Figure B.32: *Output* construct must evaluate output value.

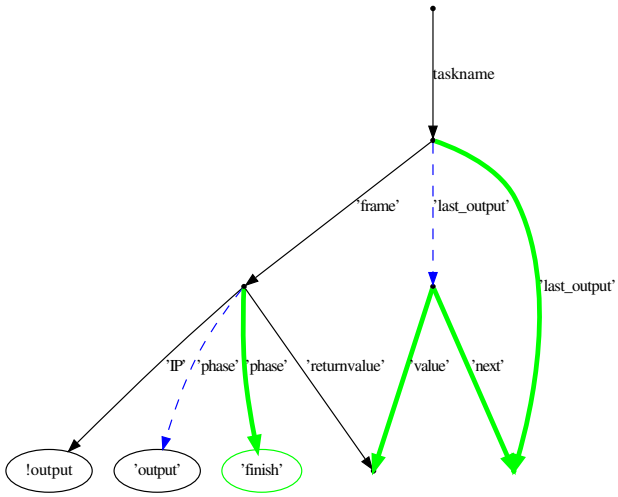


Figure B.33: *Output* construct must output the evaluated value.

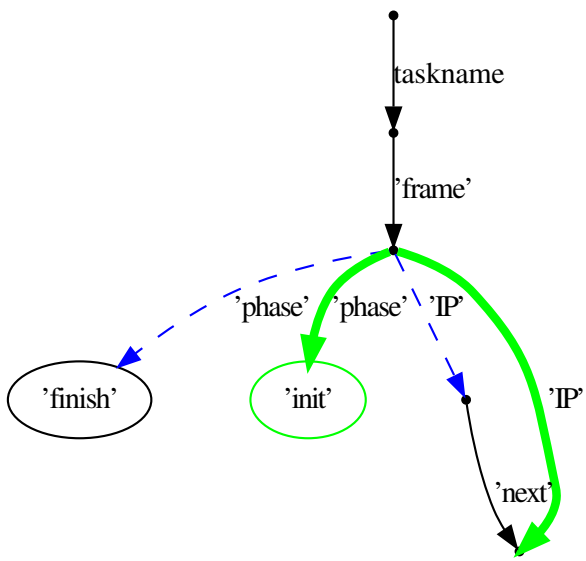


Figure B.34: Instruction has finished execution and has a next link.

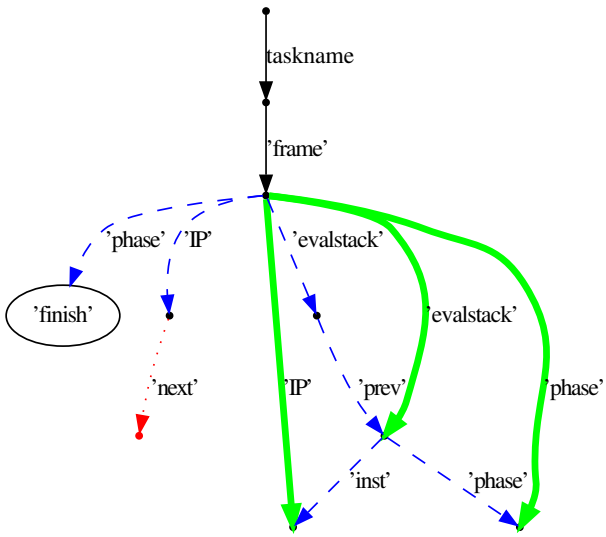


Figure B.35: Instruction has finished execution but has no next link.

phase. This popping not only sets the instruction pointer, but also copies the saved phase, making it possible to progress where we left off.

B.2 Primitives

As there are no special, built-in constructs for basic operations, such as mathematical operations, all of them have to map to a normal, user-level function. But these functions cannot implement the specified behaviour either, as the provided data values are MvS primitives. Such functions are primitive functions, which form the core of the MvK, and are hardcoded in the MvK implementation.

Primitive functions are hardcoded functions in the MvK, which get loaded like normal operations (*i.e.*, their parameters are evaluated and loaded on the stack). The execution of their body differs though, as it is executed without intermediate steps. As they cannot be written in Action Language, they do not have an implementation in the Action Language either. It is the MvK which recognizes that there is a primitive function available for the called function. If so, it calls the primitive instead of the (empty) body.

The operations in Table B.1 and B.2 need to be defined as a primitive by all Modelverse Kernel implementations, with the specified semantics. None of them are allowed to modify any of the incoming parameters. Semantics are given in simple Python code.

Name	Parameters	Returns	Semantics
integer_addition	a: Integer; b: Integer	c : Integer	$c = a + b$
integer_subtraction	a: Integer; b: Integer	c : Integer	$c = a - b$
integer_multiplication	a: Integer; b: Integer	c : Integer	$c = a \times b$
integer_division	a: Integer; b: Integer	c : Integer	$c = a/b$
integer_lt	a: Integer; b: Integer	c : Bool	$c = a < b$
integer_lte	a: Integer; b: Integer	c : Bool	$c = a \leq b$
integer_gt	a: Integer; b: Integer	c : Bool	$c = a > b$
integer_gte	a: Integer; b: Integer	c : Bool	$c = a \geq b$
integer_neg	a: Integer	c : Bool	$c = -a$
float_addition	a: Float; b: Float	c : Float	$c = a + b$
float_subtraction	a: Float; b: Float	c : Float	$c = a - b$
float_multiplication	a: Float; b: Float	c : Float	$c = a \times b$
float_division	a: Float; b: Float	c : Float	$c = a/b$
float_lt	a: Float; b: Float	c : Bool	$c = a < b$
float_lte	a: Float; b: Float	c : Bool	$c = a \leq b$
float_gt	a: Float; b: Float	c : Bool	$c = a > b$
float_gte	a: Float; b: Float	c : Bool	$c = a \geq b$
float_neg	a: Float	c : Bool	$c = -a$
bool_and	a: Bool; b: Bool	c : Bool	$c = a \wedge b$
bool_or	a: Bool; b: Bool	c : Bool	$c = a \vee b$
bool_not	a: Bool	c : Bool	$c = \neg a$
list_read	a: Element; b: Integer	c : Element	$c = a[b]$
list_append	a: Element; b: Element	a : Element	$a+ = b$
list_insert	a: Element; b: Element; c: Integer	a : Element	$a.insert(b, c)$
list_delete	a: Element; b: Integer	a : Element	$a = a.pop(b)$
list_len	a: Element	b : Integer	$b = len(a)$
dict_add	a: Element; b: Element, c: Element	a : Element	$a[b] = c$
dict_delete	a: Element; b: Element	a : Element	$delete\ a[b]$
dict_read	a: Element; b: Element	c : Element	$c = a[b.value]$
dict_read_edge	a: Element; b: Element	c : Element	$c = a[b]$
dict_read_node	a: Element; b: Element	c : Element	$c = a[b.id].edge$
dict_len	a: Element	b : Integer	$b = len(a)$
dict_in	a: Element; b: Element	c : Boolean	$c = bina$
dict_in_node	a: Element; b: Element	c : Boolean	$c = bina$
dict_keys	a: Element	b : Element	$b = a.keys()$
string_join	a: String; b: String	c : String	$c = a.b$
string_get	a: String; b: Integer	c : String	$c = a[b]$
string_split	a: String; b: String	c : Element	$c = a.split(b)$
string_len	a: String	b : Integer	$b = len(a)$
set_add	a: Element; b: Element	a : Element	$a.add(b)$
set_pop	a: Element	b : Element	$b = a.pop()$
set_remove	a: Element; b: Element	a : Element	$a.remove(b)$
set_remove_node	a: Element; b: Element	a : Element	$a.remove(b.id)$
set_in	a: Element; b: Element	c : Boolean	$c = bina$
value_eq	a: Element; b: Element	c : Bool	$c = a.value == b.value$
value_neq	a: Element; b: Element	c : Bool	$c = a.value \neq b.value$
element_eq	a: Element; b: Element	c : Bool	$c = a.id == b.id$
element_neq	a: Element; b: Element	c : Bool	$c = a.id \neq b.id$

Table B.1: Primitive functions modifying primitive datavalues. If a Value is taken or returned, this refers to the value of the returned node.

Name	Parameters	Returns	Semantics
cast_float	a : Element	b : Float	$b = float(a)$
cast_string	a : Element	b : String	$b = str(a)$
cast_integer	a : Element	b : Integer	$b = bool(a)$
cast_boolean	a : Element	b : Bool	$b = bool(a)$
cast_value	a : Element	b : String	$b = str(a.value)$
cast_id	a : Element	b : String	$b = str(a)$
create_node	—	a : Element	create node and return ID
create_edge	a : Element; b : Element	c : Edge	create edge from a to b and return ID
create_value	a : Value	b : Element	create node with value a and return ID
is_edge	a : Element	b : Boolean	return whether a is an edge or not
read_nr_out	a : Element	b : Integer	return number of outgoing links from a
read_out	a : Element; b : Integer	c : Element	return the bth outgoing link from a
read_nr_in	a : Element	b : Integer	return number of incoming links from a
read_in	a : Element; b : Integer	c : Element	return the bth incoming link from a
read_edge_src	a : Edge	b : Element	return the source of edge a
read_edge_dst	a : Edge	b : Element	return the destination of edge a
delete_element	a : Element	a : Boolean	delete element a
log	a : String	a : String	print to console at Modelverse server
is_physical_int	a : Element	b : Boolean	$type(a.value) == integer$
is_physical_float	a : Element	b : Boolean	$type(a.value) == float$
is_physical_string	a : Element	b : Boolean	$type(a.value) == string$
is_physical_boolean	a : Element	b : Boolean	$type(a.value) == boolean$
is_physical_action	a : Element	b : Boolean	$type(a.value) == action$

Table B.2: Lower-level primitive functions to implement. If a Value is taken or returned, this refers to the value of the returned node.

Bibliography

[1] Software & Systems Process Engineering Metamodel Specification. https://www.omg.org/spec/SPEM/, 2008. Cited on page 42.

[2] OMG BPMN. http://www.omg.org/spec/BPMN/, 2013. Cited on pages 42, 44, and 52.

[3] OCL. http://www.omg.org/spec/OCL/, 2014. Cited on pages 35, 41, and 98.

[4] MOF. http://www.omg.org/spec/MOF/, 2015. Cited on page 36.

[5] UML. http://www.omg.org/spec/UML/, 2015. Cited on pages 36 and 89.

[6] AnyLogic. https://anylogic.com, 2018. Cited on pages 43 and 44.

[7] AnyLogic Cloud. https://cloud.anylogic.com/, 2018. Cited on page 44.

[8] Ecore. https://wiki.eclipse.org/Ecore, 2018. Cited on page 45.

[9] ACRETOAIE, V., STÖRRLE, H., AND STRÜBER, D. Transparent model transformation: Turning your favourite model editor into a transformation tool. *Lecture Notes in Computer Science* (2015), 121–130. Cited on page 40.

[10] AL-ZOUBI, K., AND WAINER, G. Interfacing and coordination for a DEVS simulation protocol standard. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications* (2008), pp. 300–307. Cited on page 29.

[11] ALBERTSSON, L., AND MAGNUSSON, P. S. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2000), pp. 191–198. Cited on page 219.

[12] ALLEN, N. A., SHAFFER, C. A., AND WATSON, L. T. Building modeling tools that support verification, validation, and testing for the domain expert. In *Proceedings of the 37th Winter Simulation Conference* (2005), WSC '05, Winter Simulation Conference, pp. 419–426. Cited on page 215.

[13] ALTINTAS, I., BERKLEY, C., JAEGER, E., JONES, M., LUDÄSCHER, B., AND MOCK, S. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management* (2004), pp. 423–424. Cited on page 192.

[14] ÁLVAREZ, J. M., EVANS, A., AND SAMMUT, P. Mapping between levels in the metamodel architecture. In *Proceedings of the Conference on the Unified Modeling Language (UML)* (2001), pp. 34 – 46. Cited on pages 36, 90, and 93.

[15] AMÁLIO, N., DE LARA, J., AND GUERRA, E. FRAGMENTA: a theory of fragmentation for MDE. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2015), pp. 106 – 115. Cited on pages 90, 93, and 100.

[16] AMRANI, M. A set-theoretic formal specification of the semantics of kermeta. Tech. Rep. TR-LASSY-12-12, University of Luxembourg, 2012. Cited on page 41.

[17] AMRANI, M. A set-theoretic formal specification of the semantics of Kermeta. Tech. rep., University of Luxembourg, 2012. Cited on page 90.

[18] ARMSTRONG, J. The development of Erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 1997), ICFP '97, ACM, pp. 196–203. Cited on page 186.

[19] ASIKAINEN, T., AND MÄNNISTÖ, T. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling (SoSyM) 8*, 4 (2009), 521 – 549. Cited on pages 36, 90, 93, and 98.

[20] ATKINSON, C. Meta-modeling for distributed object environments. In *Proceedings of the International Workshop on Enterprise Distributed Object Computing (EDOC)* (1997), pp. 90 – 101. Cited on page 40.

[21] ATKINSON, C., AND GERBIG, R. Melanie: multi-level modeling and ontology engineering environment. In *Proceedings of the Master Class on Model-Driven Engineering: Modeling Wizards* (2012), pp. 7:1 – 7:2. Cited on page 98.

[22] ATKINSON, C., AND GERBIG, R. Aspect-oriented concrete syntax definition for deep modeling languages. In *Proceedings of the Workshop on Multi-Level Modelling (MULTI)* (2015), pp. 13 – 22. Cited on pages 93 and 98.

[23] ATKINSON, C., GERBIG, R., AND KÜHNE, T. Comparing multi-level modeling approaches. In *Proceedings of the Workshop on Multi-Level Modelling (MULTI)* (2014), pp. 53 – 61. Cited on pages 39, 40, and 98.

[24] ATKINSON, C., GERBIG, R., AND KÜHNE, T. Opportunities and challenges for deep constraint languages. In *Proceedings of the Workshop on OCL and Textual Modelling (OCL)* (2015), pp. 3 – 18. Cited on pages 41 and 98.

[25] ATKINSON, C., GERBIG, R., AND KÜHNE, T. A unifying approach to connections for multi-level modeling. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2015), pp. 216 – 225. Cited on pages 39, 90, and 98.

[26] ATKINSON, C., GERBIG, R., AND METZGER, N. On the execution of execution of deep models. In *Proceedings of the International Workshop on Executable Modeling (EXE)* (2015), pp. 28 – 33. Cited on page 39.

[27] ATKINSON, C., KENNEL, B., AND GOSS, B. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *Proceedings of the Workshop on Semantic Web Enabled Software Engineering (SWESE)* (2011), pp. 1:1 – 1:15. Cited on pages 37, 98, and 108.

[28] ATKINSON, C., AND KÜHNE, T. Strict profiles: Why and how. *Lecture Notes in Computer Science 1939* (2000), 309–322. Cited on page 109.

[29] ATKINSON, C., AND KÜHNE, T. The essence of multilevel metamodeling. In *Proceedings of the Conference on the Unified Modeling Language (UML)* (2001),

pp. 19 – 33. Cited on pages 89 and 98.

[30] ATKINSON, C., AND KÜHNE, T. Profiles in a strict metamodeling framework. *Science of Computer Programming 44*, 1 (2002), 5 – 22. Cited on page 90.

[31] ATKINSON, C., AND KÜHNE, T. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 12*, 4 (2002), 290 – 321. Cited on pages 38, 39, and 90.

[32] ATKINSON, C., AND KÜHNE, T. Concepts for comparing modeling tool architectures. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2005), pp. 398 – 413. Cited on pages 35, 38, 89, 97, 108, and 109.

[33] ATKINSON, C., AND KÜHNE, T. Reducing accidental complexity in domain models. *Software and Systems Modeling (SoSyM) 7*, 3 (2008), 345 – 359. Cited on page 39.

[34] BAKER, P., LOH, S., AND WEIL, F. Model-driven engineering in a large industrial context — motorola case study. *Lecture Notes in Computer Science* (2005), 476–491. Cited on pages 2 and 3.

[35] BARROCA, B., KÜHNE, T., AND VANGHELUWE, H. Integrating language and ontology engineering. In *Proceedings of the Workshop on Multi-Paradigm Modeling (MPM)* (2014), pp. 77 – 86. Cited on page 38.

[36] BASCIANI, F., DI ROCCO, J., DI RUSCIO, D., DI SALLE, A., IOVINO, L., AND PIERANTONIO, A. MDEForge: an extensible web-based modeling platform. In *Proceedings of the Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)* (2014), pp. 66 – 75. Cited on pages 44, 45, 56, 90, and 98.

[37] BENDRAOU, R., COMBEMALE, B., CRÉGUT, X., AND GERVAIS, M.-P. Definition of an eXecutable SPEM. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)* (2007), pp. 390–397. Cited on page 42.

[38] BENDRAOU, R., JEZEQUEL, J.-M., GERVAIS, M.-P., AND BLANC, X. A comparison of six UML-based languages for software process modeling. *IEEE Transactions on Software Engineering 36*, 5 (2010), 662–675. Cited on page 42.

[39] BERGERO, F., AND KOFMAN, E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation 87* (2011), 113–132. Cited on pages 29 and 144.

[40] BERNSTEIN, P. A. Applying model management to classical meta data problems. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)* (2003), pp. 19:1 – 19:12. Cited on page 90.

[41] BÉZIVIN, J. On the unification power of models. *Software and Systems Modeling (SoSyM) 4*, 2 (2005), 171 – 188. Cited on pages 42, 89, and 90.

[42] BÉZIVIN, J., JOUAULT, F., ROSENTHAL, P., AND VALDURIEZ, P. Modeling in the large and modeling in the small. In *Proceedings of the European Conference on Model Driven Architecture: Foundations and Applications (ECMFA)* (2005), pp. 33 – 46. Cited on pages 3 and 42.

[43] BÉZIVIN, J., JOUAULT, F., ROSENTHAL, P., AND VALDURIEZ, P. Modeling in the large and modeling in the small. In *Proceedings of the European Conference on Model Driven Architecture: Foundations and Applications (ECMFA)* (2005), pp. 33 – 46. Cited on pages 35, 43, and 100.

[44] BÉZIVIN, J., JOUAULT, F., AND VALDURIEZ, P. On the need for megamodels.

In *Proceedings of the Workshop on Best Practices for Model Driven Software Development (MDSD)* (2004), pp. 1:1 – 1:9. Cited on pages 43 and 100.

[45] BHATTACHARJEE, A. K., AND SHYAMASUNDAR, R. K. Activity diagrams : A formal framework to model business processes and code generation. *Journal of Object Technology 8*, 1 (2009), 189–220. Cited on page 121.

[46] BIEHL, M., EL-KHOURY, J., LOIRET, F., AND TÖRNGREN, M. On the modeling and generation of service-oriented tool chains. *Software & Systems Modeling 13*, 2 (2014), 461–480. Cited on page 123.

[47] BLOUIN, D., SENN, E., ROUSSEL, K., AND ZENDRA, O. QAML: A multi-paradigm DSML for quantitative analysis of embedded system architecture models. In *Proceedings of the International Workshop on Multi-Paradigm Modeling* (2012), pp. 37–42. Cited on pages 2 and 4.

[48] BONAVENTURA, M., WAINER, G., AND CASTRO, R. Graphical modeling and simulation of discrete-event systems with CD++Builder. *SIMULATION 89*, 1 (2013), 4–27. Cited on page 143.

[49] BOOTHE, B. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (2000), PLDI '00, pp. 299–310. Cited on page 216.

[50] BOUSSE, E., CORLEY, J., COMBEMALE, B., GRAY, J., AND BAUDRY, B. Supporting efficient and advanced omniscient debugging for xdsmls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (2015), SLE 2015, pp. 137–148. Cited on pages 164, 187, 209, 215, and 217.

[51] BOUSSE, E., MAYERHOFER, T., COMBEMALE, B., AND BAUDRY, B. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *11th European Conference on Modelling Foundations and Applications (ECMFA)* (L'Aquila, Italy, 2015). Cited on page 187.

[52] BROMAN, D., LEE, E. A., TRIPAKIS, S., AND TÖRNGREN, M. Viewpoints, formalisms, languages, and tools for cyber-physical systems. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling* (2012), ACM, pp. 49–54. Cited on page 1.

[53] BROSCH, P., KAPPEL, G., LANGER, P., SEIDL, M., WIELAND, K., AND WIMMER, M. An introduction to model versioning. In *Formal Methods for Model-Driven Engineering - International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)* (2012), pp. 336 – 398. Cited on page 56.

[54] BRUNET, G., CHECHIK, M., EASTERBROOK, S., NEJATI, S., NIU, N., AND SABETZADEH, M. A manifesto for model merging. In *Proceedings of the International Workshop on Global integrated Model Management* (2006), pp. 5 – 12. Cited on pages 81 and 176.

[55] BURCKHARDT, S., FÄHNDRICH, M., AND KATO, J. It's alive! continuous feedback in UI programming. In *Proceedings of PLDI '13* (2013), pp. 95–104. Cited on pages 165 and 187.

[56] BURDEN, H., HELDAL, R., AND LUNDQVIST, M. Industrial experiences from multi-paradigmatic modelling of signal processing. In *Proceedings of International Workshop on Multi-Paradigm Modeling* (2012), pp. 7–12. Cited on page 2.

[57] BURNETT, M. M., ATWOOD, JR., J. W., AND WELCH, Z. T. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings of Visual Languages '98* (1998), pp. 126–133. Cited on page 187.

[58] CAPOCCHI, L., SANTUCCI, J. F., POGGI, B., AND NICOLAI, C. DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems. In *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises* (2011), pp. 170–175. Cited on page 144.

[59] CARDOEN, B., MANHAEVE, S., TUIJN, T., VAN TENDELOO, Y., VANMECHELEN, K., VANGHELUWE, H., AND BROECKHOVE, J. Performance analysis of a PDEVS simulator supporting multiple synchronization protocols. In *Proceedings of the 2016 Symposium on Theory of Modeling and Simulation - DEVS* (Apr. 2016), TMS/DEVS '16, part of the Spring Simulation Multi-Conference, Society for Computer Simulation International, pp. 614 – 621. Cited on page 116.

[60] CARDOEN, B., MANHAEVE, S., VAN TENDELOO, Y., AND BROECKHOVE, J. A PDEVS simulator supporting multiple synchronization protocols: implementation and performance analysis. *SIMULATION 93* (2017). Cited on page 116.

[61] CELLIER, F. E. *Continuous System Modeling*, first ed. Springer-Verlag, 1991. Cited on pages 14, 21, 23, 165, 169, 170, 189, and 191.

[62] CHEZZI, C. M., TYMOSCHUK, A. R., AND LERMAN, R. A Method for DEVS Simulation of e-Commerce Processes for Integrated Business and Technology Evaluation (WIP). In *Proceedings of the 2013 Spring Simulation Multiconference* (2013), pp. 13:1–13:6. Cited on page 144.

[63] CHIŞ, A., DENKER, M., GÎRBA, T., AND NIERSTRASZ, O. Practical domain-specific debuggers using the moldable debugger framework. *Computer Languages, Systems & Structures 44*, PA (2015), 89–113. Cited on pages 164 and 215.

[64] CHOU, S.-C. A process modeling language consisting of high level UML-based diagrams and low level process language. *The Journal of Object Technology 1*, 4 (2002), 137–163. Cited on page 42.

[65] CHOW, A. C. H., AND ZEIGLER, B. P. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 1994 Winter Simulation Multiconference* (1994), pp. 716–722. Cited on page 143.

[66] CLARK, T., GONZALEZ-PEREZ, C., AND HENDERSON-SELLERS, B. A foundation for multi-level modelling. In *Proceedings of the Workshop on Multi-Level Modelling (MULTI)* (2014), pp. 43 – 52. Cited on pages 40 and 109.

[67] CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND QUESADA, J. Maude: specification and programming in rewriting logic. *Theoretical Computer Science 285*, 28 (2002), 187 – 243. Cited on pages 90 and 93.

[68] CLEARY, J., GOMES, F., UNGER, B., XIAO, Z., AND THUDT, R. Cost of state saving & rollback. *SIGSIM Simululation Digest 24*, 1 (1994), 94–101. Cited on pages 211 and 217.

[69] CORLEY, J. Debugging for model transformations. In *Proceedings of the MODELS 2013 Doctoral Symposium co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, October 1, 2013.* (2013), pp. 17–24. Cited on page 215.

[70] CORLEY, J., EDDY, B. P., AND GRAY, J. Towards efficient and scalabale omniscient debugging for model transformations. In *Proceedings of the 14th Workshop on Domain-Specific Modeling* (2014), DSM '14, ACM, pp. 13–18. Cited on page 216.

[71] CORLEY, J., EDDY, B. P., SYRIANI, E., AND GRAY, J. Efficient and scalable omniscient debugging for model transformations. *Software Quality Journal 25* (2017), 7–48. Cited on pages 209 and 216.

[72] COSTAGLIOLA, G., DEUFEMIA, V., AND POLESE, G. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Transactions on Software Engineering Methodology 13*, 4 (2004), 431–487. Cited on page 13.

[73] CUMBERLIDGE, M. *Business process management with JBoss jBPM*. Packt Publishing Ltd, 2007. Cited on page 122.

[74] CZAPLICKI, E. Elm: Concurrent FRP for functional GUIs. `https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf`, 2012. Cited on pages 165, 167, and 186.

[75] DÁVID, I., DENIL, J., GADEYNE, K., AND VANGHELUWE, H. Engineering process transformation to manage (in)consistency. In *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016)* (2016), http://ceur-ws.org/Vol-1717/, pp. 7–16. Cited on pages 56 and 128.

[76] DÁVID, I., DENIL, J., AND VANGHELUWE, H. Towards inconsistency management by process-oriented dependency modeling. In *International Workshop on Collaborative Modelling in MDE* (2016), pp. 35–44. Cited on pages 5 and 42.

[77] DAVIS, R. Magic paper: Sketch-understanding research. *Computer* (2007), 34–41. Cited on page 196.

[78] DE LARA, J., DI ROCCO, J., DI RUSCIO, D., GUERRA, E., IOVINO, L., PIERANTONIO, A., AND CUADRADO, J. S. Reusing model transformations through typing requirements models. *Lecture Notes in Computer Science 10202* (2017), 264–282. Cited on page 40.

[79] DE LARA, J., AND GUERRA, E. Deep meta-modelling with MetaDepth. In *Proceedings of the TOOLS EUROPE Conference* (2010), pp. 1 – 20. Cited on pages 44, 45, 89, 91, 98, and 109.

[80] DE LARA, J., AND GUERRA, E. Generic meta-modelling with concepts, templates and mixin layers. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2010), pp. 16 – 30. Cited on pages 93, 98, and 108.

[81] DE LARA, J., GUERRA, E., COBOS, R., AND LLORENA, J. M. Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal 57*, 1 (2014), 36 – 58. Cited on pages 39, 90, 92, and 98.

[82] DE LARA, J., GUERRA, E., AND SÁNCHEZ CUADRADO, J. When and how to use multilevel modelling. *ACM Transactions on Software Engineering and Methodology (TOSEM) 24*, 2 (2014), 12:1 – 12:46. Cited on pages 39, 40, 93, and 98.

[83] DE LARA, J., GUERRA, E., AND SÁNCHEZ CUADRADO, J. A-posteriori typing for model-driven engineering. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2015), pp. 156 – 165. Cited on pages 38, 98, and 108.

[84] DE LARA, J., GUERRA, E., AND SÁNCHEZ CUADRADO, J. Model-driven engineering with domain-specific meta-modelling languages. *Software and Systems Modeling (SoSyM) 14*, 1 (2015), 429 – 459. Cited on pages 45, 89, 98, and 205.

[85] DE LARA, J., GUERRA, E., AND VANGHELUWE, H. A multi-view component modelling language for systems design: Checking consistency and timing constraints. In *Visual Modeling for Software Intensive Systems* (2005), pp. 27–34. Cited on page 204.

[86] DE LARA, J., AND VANGHELUWE, H. Atom3: A tool for multi-formalism and meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering* (2002), pp. 174–188. Cited on pages 43, 89, 109, and 203.

[87] DEBRECENI, C., BERGMANN, G., RÁTH, I., AND VARRÓ, D. Property-based locking in collaborative modeling. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems* (2017), pp. 199–209. Cited on page 153.

[88] DEGUEULE, T., COMBEMALE, B., BLOUIN, A., BARAIS, O., AND JÉZÉQUEL, J.-M. Melange: a meta-language for modular and reusable development of DSLs. In *Proceedings of the International Conference on Software Language Engineering (SLE)* (2015), pp. 25 – 36. Cited on page 90.

[89] DEGUEULE, T., COMBEMALE, B., BLOUIN, A., BARAIS, O., AND JÉZÉQUEL, J.-M. Safe model polymorphism for flexible modeling. *Computer Languages, Systems & Structures 49* (2017), 176–195. Cited on pages 89 and 97.

[90] DEMUTH, A., RIEDL-EHRENLEITNER, M., AND EGYED, A. Towards flexible, incremental, and paradigm-agnostic consistency checking in multi-level modeling environments. In *Proceedings of the Workshop on Multi-Level Modelling (MULTI)* (2014), pp. 73 – 82. Cited on pages 90, 92, and 98.

[91] DENIL, J. *Design, Verification and Deployment of Software-Intensive Systems: A Multi-Paradigm Modelling Approach*. PhD thesis, University of Antwerp, 2013. Cited on pages 33 and 50.

[92] DENIL, J., JUKŠS, M., VERBRUGGE, C., AND VANGHELUWE, H. Search-based model optimization using model transformations. In *Proceedings of the International Conference on System Analysis and Modeling* (2014), pp. 80–95. Cited on page 41.

[93] DENIL, J., SALAY, R., PAREDIS, C., AND VANGHELUWE, H. Towards agile model-based systems engineering. In *Proceedings of MODELS 2017 Satellite Event* (2017), pp. 424–429. Cited on page 4.

[94] DESHAYES, R., JACQUET, C., HARDEBOLLE, C., BOULANGER, F., AND MENS, T. Heterogeneous modeling of gesture-based 3D applications. In *Proceedings of the International Workshop on MPM* (2012), pp. 19–24. Cited on page 2.

[95] DESHAYES, R., MEYERS, B., MENS, T., AND VANGHELUWE, H. ProMoBox in practice: A case study on the GISMO domain-specific modelling language. In *Proceedings of MPM* (2014), pp. 21–30. Cited on page 5.

[96] DÉVA, G., KOVÁCS, G. F., AND AN, A. Textual, executable, translatable UML. In *Proceedings of the Workshop on OCL and Textual Modelling (OCL)* (2014), pp. 3 – 12. Cited on pages 92 and 164.

[97] DI ROCCO, J., DI RUSCIO, D., IOVINO, L., AND PIERANTONIO, A. Collaborative repositories in model-driven engineering. *IEEE Software* (2015), 28–34. Cited on

pages 43 and 52.

[98] DI ROCCO, J., DI RUSCIO, D., IOVINO, L., AND PIERANTONIO, A. Collaborative repositories in model-driven engineering. *IEEE Software 32*, 3 (2015), 28 – 34. Cited on page 90.

[99] DI RUCCO, J., DI RUSCIO, D., PIERANTOINI, A., SÁNCHEZ CUADRADO, J., DE LARA, J., AND GUERRA, E. Using ATL transformation services in the MDE-Forge collaborative modeling platform. In *Proceedings of the International Conference on Model Transformation (ICMT)* (2016). Cited on pages 45, 89, 90, and 98.

[100] DI SANDRO, A., SALAY, R., FAMELIS, M., KOKALY, S., AND CHECHIK, M. MMINT: a graphical tool for interactive model management. In *Proceedings of the MoDELS Demo and Poster Session* (2015), pp. 16 – 19. Cited on pages 42, 44, 45, 56, 90, and 109.

[101] DUBÉ, D. Graph layout for domain-specific modeling. Master's thesis, McGill University, 2006. Cited on page 189.

[102] EDWARDS, J. Subtext: Uncovering the simplicity of programming. In *Proceedings of OOPSLA '05* (2005), pp. 505–518. Cited on page 186.

[103] ENGBLOM, J. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference* (2012), pp. 1–6. Cited on page 216.

[104] ERNST, J. Data interoperability between CACSD and CASE tools using the CDIF family of standards. In *Proceedings of the 1996 International Symposium on Computer Aided Control System Design* (1996), pp. 346–351. Cited on page 3.

[105] ETZLSTORFER, J., KUSEL, A., KAPSAMMER, E., LANGER, P., RETSCHITZEGGER, W., SCHOENBOECK, J., SCHWINGER, W., AND WIMMER, M. A survey on incremental model transformation approaches. In *Proceedings of the Workshop on Models and Evolution* (2013), pp. 4–13. Cited on page 41.

[106] EYSHOLDT, M., AND BEHRENS, H. Xtext: implement your language faster than the quick and dirty way. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA)* (2010), pp. 307–309. Cited on page 204.

[107] FABRY, R. S. How to design a system in which modules can be changed on the fly. In *Proceedings of ICSE '76* (1976), pp. 470–476. Cited on pages 187 and 188.

[108] FAVRE, J.-M. Languages evolve too! changing the software time scale. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution* (Washington, DC, USA, 2005), IWPSE '05, IEEE Computer Society, pp. 33–44. Cited on page 188.

[109] FAVRE, J.-M. Megamodelling and etymology. In *Proceedings of the Dagstuhl Seminar on Transformation Techniques in Software Engineering* (2006), pp. 5:1 – 5:22. Cited on page 93.

[110] FERAYORNI, A. E., AND SARJOUGHIAN, H. S. Domain driven simulation modeling for software design. In *Proceedings of the 2007 Summer Computer Simulation Conference* (2007), pp. 297–304. Cited on page 144.

[111] FOSTER, H., UCHITEL, S., MAGEE, J., AND KRAMER, J. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada* (2003),

pp. 152–163. Cited on page 122.

[112] FRANCE, R., BIEMAND, J., AND CHENG, B. H. C. Repository for model driven development. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2006), pp. 311 – 317. Cited on pages 44, 46, and 98.

[113] FRANCE, R., BIEMAND, J., AND CHENG, B. H. C. Repository for model driven development. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2006), pp. 311 – 317. Cited on page 90.

[114] FU, X., BULTAN, T., AND SU, J. Analysis of interacting BPEL web services. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004* (2004), pp. 621–630. Cited on page 122.

[115] FUJIMOTO, R. M. Parallel discrete event simulation. *Communications of the ACM* (1990), 30–53. Cited on pages 115 and 217.

[116] FUJIMOTO, R. M. Performance of time warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990). Cited on page 115.

[117] FUJIMOTO, R. M. *Parallel and Distribution Simulation Systems*, 1st ed. John Wiley & Sons, Inc., 1999. Cited on pages 115, 206, 211, and 217.

[118] GALLARDO, J., BRAVO, C., AND REDONDO, M. A. A model-driven development method for collaborative modeling tools. *Journal of Network and Computer Applications 35* (2012), 1086–1105. Cited on page 43.

[119] GARLAN, D., MONROE, R. T., AND WILE, D. Acme: An architecture description interchange language. In *Proceedings of CASCON'97* (1997), pp. 169–183. Cited on page 3.

[120] GDB. GDB reversible debugging. `https://www.gnu.org/software/gdb/news/reversible.html`, 2009. Cited on page 208.

[121] GÉRARD, S. Once upon a time, there was Papyrus... In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (2015), pp. IS–7. Cited on page 204.

[122] GITZEL, R., OTT, I., AND SCHADER, M. Ontological extension to the MOF metamodel as a basis for code generation. *The Computer Journal 50*, 1 (2007), 93–115. Cited on page 63.

[123] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983. Cited on pages 167 and 186.

[124] GOLDSTEIN, R., AND KHAN, A. A taxonomy of event time representations. In *Proceedings of the Spring Simulation Conference* (2017), pp. 6:1–6:12. Cited on pages 143 and 206.

[125] GÓMEZ, A., MENDIALDUA, X., BERGMANN, G., CABOT, J., DEBRECENI, C., GARMENDIA, A., KOLOVOS, D. S., DE LARA, J., AND TRUJILLO, S. On the opportunities of scalable modeling technologies: An experience report on wind turbines control applications development. *Lecture Notes in Computer Science 10376* (2017), 300 – 315. Cited on pages 3, 4, and 43.

[126] GONZALEZ-PEREZ, C., AND HENDERSON-SELLERS, B. A powertype-based metamodelling framework. *Software and Systems Modeling (SoSyM) 5*, 1 (2006), 72 – 90. Cited on page 39.

[127] GRAY, J. Why do computers stop and what can be done about it? Tech. rep., Tandem Computers, 1985. Cited on pages 206 and 219.

[128] GRÖNNIGER, H., KRAHN, H., RUMPE, B., SCHINDLER, M., AND VÖLKEL, S. Text-based modeling. In *Proceedings of the 4th International Workshop on Software Language Engineering* (2007). Cited on pages 54, 69, and 186.

[129] GROOTHUIS, M., FRIJNS, R., VOETEN, J., AND BROENINK, J. Concurrent design of embedded control software. *ECEASST* (2009). Cited on pages 2 and 4.

[130] GUERRA, E., AND DE LARA, J. Towards automating the analysis of integrity constraints in multi-level models. In *Proceedings of the Workshop on Multi-Level Modelling (MULTI)* (2014), pp. 1 – 10. Cited on pages 37 and 98.

[131] GUY, C., COMBEMALE, B., DERRIEN, S., STEEL, J. R. H., AND JÉZÉQUEL, J.-M. On model subtyping. In *Proceedings of the European Conference on Model Driven Architecture: Foundations and Applications (ECMFA)* (2012), pp. 400 – 415. Cited on page 89.

[132] HANCOCK, C. M. *Real-Time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, 2003. Cited on page 186.

[133] HAREL, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8*, 3 (1987), 231–274. Cited on pages 14, 15, 119, 122, and 208.

[134] HAREL, D., AND GERY, E. Executable object modeling with statecharts. *IEEE Computer 30*, 7 (1997), 31 – 42. Cited on page 93.

[135] HAREL, D., AND NAAMAD, A. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering Methodology 5*, 4 (1996), 293–333. Cited on page 15.

[136] HAREL, D., AND RUMPE, B. Meaningful modeling: What's the semantics of "semantics"? *Computer 37*, 10 (2004), 64–72. Cited on page 11.

[137] HENDERSON-SELLERS, B. Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software 84*, 2 (2011), 301 – 313. Cited on page 38.

[138] HENDERSON-SELLERS, B., CLARK, T., AND GONZALEZ-PEREZ, C. On the search for a level-agnostic modelling language. In *Proceedings of the International Conference on Advanced Information Systems Engineering* (2013), pp. 240 – 255. Cited on pages 40 and 109.

[139] HENDERSON-SELLERS, B., AND GONZALEZ-PEREZ, C. A comparison of four process metamodels and the creation of a new generic standard. *Information and Software Technology 47*, 1 (2005), 49–65. Cited on page 42.

[140] HENKLER, S., AND HIRSCH, M. A multi-paradigm modeling approach for reconfigurable mechatronic systems. In *Proceedings of the International Workshop on MPM* (2006), pp. 15–25. Cited on page 3.

[141] HEROLD, S. Compliance between architecture and design models of component-based systems. *ECEASST* (2010). Cited on page 4.

[142] HERRMANNSDÖRFER, M., AND HUMMEL, B. Library concepts for model reuse.

In *Proceedings of the Workshop on Language Descriptions Tools and Applications (LDTA)* (2009), pp. 121 – 134. Cited on page 90.

[143] HOPCROFT, J. E., MOTWANI, R., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. Cited on pages 20, 165, and 169.

[144] HWANG, M. H. X-S-Y. `https://code.google.com/p/x-s-y/`, 2012. Cited on pages 29 and 144.

[145] IUGAN, L. G., NICOLESCU, G., AND O'CONNOR, I. Modeling and formal verification of a passive optical network on chip behaviour. *ECEASST* (2009). Cited on page 3.

[146] JEFFERSON, D. R. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3 (1985), 404–425. Cited on page 209.

[147] JIANG, J., AND SYSTÄ, T. A pattern-based approach to manage model references. *ECEASST* (2009). Cited on pages 4 and 5.

[148] JOHNSON, G., GROSS, M. D., HONG, J., AND YI-LUEN DO, E. Computational support for sketching in design: a review. *Foundations and Trends in Human-Computer Interaction 2*, 1 (2009), 1 – 93. Cited on page 37.

[149] JOUAULT, F., ALLILAIRE, F., BÉZIVIN, J., AND KURTEV, I. ATL: A model transformation tool. *Science of Computer Programming 72*, 1-2 (2008), 31–39. Cited on page 45.

[150] JUKŠS, M., VERBRUGGE, C., ELAASAR, M., AND VANGHELUWE, H. Scope in model transformations. *Software & Systems Modeling* (2016), 1–26. Cited on page 202.

[151] JUKŠS, M., VERBRUGGE, C., VARRÓ, D., AND VANGHELUWE, H. Dynamic scope discovery for model transformations. In *Proceedings of the International Conference on Software Language Engineering (SLE)* (2014), pp. 302–321. Cited on page 41.

[152] KANTNER, D. Specification and implementation of a deep OCL dialect. Master's thesis, University of Mannheim, 2014. Cited on page 41.

[153] KAPPEL, G., LANGER, P., RETZSCHITZEGGER, W., SCHWINGER, W., AND WIMMER, M. Model transformation by-example: A survey of the first wave. *Lecture Notes in Computer Science 7260* (2012), 197–215. Cited on page 40.

[154] KELLY, S., AND TOLVANEN, J.-P. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008. Cited on pages 9, 44, 45, 164, 187, and 204.

[155] KEMPER, P. A trace-based visual inspection technique to detect errors in simulation models. In *2007 Winter Simulation Conference* (2007), pp. 747–755. Cited on page 215.

[156] KENNEL, B. *A Unified Framework for Multi-Level Modeling*. PhD thesis, University of Mannheim, 2012. Cited on page 97.

[157] KENT, S. *Model Driven Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 286–298. Cited on page 1.

[158] KIM, S., SARJOUGHIAN, H. S., AND ELAMVAZHUTHI, V. DEVS-Suite: a simulator supporting visual experimentation design and behavior monitoring. In *Proceedings of the 2009 Spring Simulation Multiconference* (2009), pp. 161:1–161:7. Cited

on pages 29 and 143.

[159] KIM, T., LEE, C., CHRISTENSEN, E., AND ZEIGLER, B. System entity structuring and model base management. *IEEE Transactions on Systems, Man and Cybernetics 20*, 5 (Sept. 1990), 1013–1024. Cited on page 144.

[160] KITCHIN, D., QUARK, A., COOK, W. R., AND MISRA, J. The orc programming language. In *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009.* (2009), pp. 1–25. Cited on page 122.

[161] KLEPPE, A. A language description is more than a metamodel. In *Proceedings of the International Conference on Software Language Engineering (SLE)* (2007). Cited on pages 10 and 189.

[162] KOFMAN, E., LAPADULA, M., AND PAGLIERO, E. PowerDEVS: A DEVS-Based Environment for Hybrid System Modeling and Simulation. Tech. rep., School of Electronic Engineering, Universidad Nacional de Rosario, 2003. Cited on page 144.

[163] KOLOVOS, D. S. Establishing correspondences between models with the epsilon comparison language. *Lecture Notes in Computer Science 5562* (2009), 146–157. Cited on pages 41 and 81.

[164] KOLOVOS, D. S., PAIGE, R. F., AND POLACK, F. A. C. The Epsilon Object Language (EOL). In *Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications (ECMFA)* (2006), pp. 128 – 142. Cited on pages 41 and 98.

[165] KOLOVOS, D. S., PAIGE, R. F., AND POLACK, F. A. C. Merging models with the Epsilon Merging Language (EML). *Lecture Notes in Computer Science 4199* (2006), 215–229. Cited on pages 41 and 81.

[166] KOLOVOS, D. S., PAIGE, R. F., AND POLACK, F. A. C. The Epsilon Transformation Language. *Lecture Notes in Computer Science 5063* (2008), 46–60. Cited on page 45.

[167] KOVÁCS, M., VARRÓ, D., AND GÖNCZY, L. Formal analysis of BPEL workflows with compensation by model checking. *Comput. Syst. Sci. Eng. 23*, 5 (2008). Cited on page 122.

[168] KUHN, A., MURPHY, G. C., AND THOMPSON, C. A. An exploratory study of forces and frictions affecting large-scale model-driven development. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems* (Berlin, Heidelberg, 2012), MODELS'12, Springer-Verlag, pp. 352–367. Cited on page 165.

[169] KÜHNE, T. Matters of (meta-)modeling. *Software and Systems Modeling (SoSyM) 5* (2006), 369 – 385. Cited on pages 11, 35, 36, 40, 48, 90, and 187.

[170] KÜHNE, T. On model compatibility with referees and contexts. *Software and Systems Modeling (SoSyM) 12*, 3 (2013), 475 – 488. Cited on page 90.

[171] KÜHNE, T., MEZEI, G., SYRIANI, E., VANGHELUWE, H., AND WIMMER, M. Explicit transformation modeling. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2009), pp. 240 – 255. Cited on pages 40, 44, 94, and 159.

[172] KURTEV, I., BÉZIVIN, J., JOUAULT, F., AND VALDURIEZ, P. Model-based DSL

frameworks. In *Proceedings of the Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2006), pp. 602 – 616. Cited on page 43.

[173] LAMO, Y., WANG, X., MANTZ, F., MACCAULL, W., AND RUTLE, A. DPF Workbench: A diagrammatic multi-layer domain specific (meta-) modelling environment. *Computer and Information Science, Studies in Computational Intelligence 429* (2012), 37–52. Cited on pages 43 and 89.

[174] LATOMBE, F., CRÉGUT, X., COMBEMALE, B., DEANTONI, J., AND PANTEL, M. Weaving concurrency in eXecutable Domain-Specific Modeling Languages. In *Proceedings of the International Conference on Software Language Engineering (SLE)* (2015), pp. 125 – 136. Cited on pages 5, 41, and 99.

[175] LEDECZI, A., MAROTI, M., BAKAY, A., AND KARSAI, G. The Generic Modeling Environment. In *Proceedings of International Symposium on Intelligent Signal Processing (WISP)* (2001). Cited on pages 43, 89, 90, and 92.

[176] LEE, E. The problem with threads. Tech. rep., University of California at Berkeley, 2006. Cited on pages 3 and 120.

[177] LEVENDOVSZKY, T., LENGYEL, L., MEZEI, G., AND CHARAF, H. A systematic approach to metamodeling environments and model transformation systems in VMTS. In *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs)* (2004), pp. 65 – 75. Cited on pages 63 and 90.

[178] LEWIS, B. Debugging backwards in time. *arXiv preprint cs/0310016*, September (2003), 225–235. Cited on page 208.

[179] LI, X., VANGHELUWE, H., LEI, Y., SONG, H., AND WANG, W. A testing framework for DEVS formalism implementations. In *Proceedings of the 2011 Spring Simulation Multiconference* (2011), pp. 183–188. Cited on pages xxxv, 143, 145, and 146.

[180] LIEBERMAN, H., AND FRY, C. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (1995), pp. 480–486. Cited on page 165.

[181] LIENHARD, A., GÎRBA, T., AND NIERSTRASZ, O. Practical object-oriented back-in-time debugging. In *ECOOP 2008 – Object-Oriented Programming* (2008), pp. 592–615. Cited on page 209.

[182] LINDEMAN, R. T., KATS, L. C. L., AND VISSER, E. Declaratively defining domain-specific language debuggers. In *Proceedings of the 10th International Conference on Generative Programming and Component Engineering* (2011), pp. 127–136. Cited on page 164.

[183] LÓPEZ-FERNÁNDEZ, J. J., CUADRADO, J. S., GUERRA, E., AND DE LARA, J. Example-driven meta-model development. *Software and Systems Modeling 14*, 4 (2013), 1323–1347. Cited on page 49.

[184] LÚCIO, L., ABID, S. B., RAHMAN, S., ARAVANTINOS, V., KUESTNER, R., AND HARWARDT, E. Process-aware model-driven development environments. In *Proceedings of MODELS 2017 Satellite Event* (2017), pp. 405–411. Cited on page 4.

[185] LÚCIO, L., MUSTAFIZ, S., DENIL, J., MEYERS, B., AND VANGHELUWE, H. The formalism transformation graph as a guide to model driven engineering. Tech. Rep. SOCS-TR2012-1, School of Computer Science, McGill University, 2012. Cited on

pages 3, 4, and 124.

[186] LÚCIO, L., MUSTAFIZ, S., DENIL, J., VANGHELUWE, H., AND JUKŠS, M. FTG+PM: An integrated framework for investigating model transformation chains. *Lecture Notes in Computer Science 7916* (2013), 182–202. Cited on pages 31, 42, 124, and 179.

[187] MADIOT, F., AND PAGANELLI, M. Eclipse sirius demonstration. In *Proceedings of the MoDELS 2015 Demo and Poster Session* (2015), pp. 9–11. Cited on page 204.

[188] MALEKI, M. M., WOODBURY, R. F., GOLDSTEIN, R., BRESLAV, S., AND KHAN, A. Designing DEVS visual interfaces for end-user programmers. *Simulation 91*, 8 (2015), 715–734. Cited on page 215.

[189] MANNADIAR, R., AND VANGHELUWE, H. Debugging in domain-specific modelling. In *Software Language Engineering*, B. Malloy, S. Staab, and M. Brand, Eds., vol. 6563 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 276–285. Cited on pages 187 and 215.

[190] MANZANARES, C. C., CUADRADO, J. S., AND DE LARA, J. Building MDE cloud services with DISTIL. In *Proceedings of the International Workshop on Model-Driven Engineering on and for the Cloud* (2015), pp. 19–24. Cited on pages 90 and 109.

[191] MARÓTI, M., KECSKÉS, T., KERESKÉNYI, R., BROLL, B., VÖLGYESI, P., JURÁCZ, L., LEVENDOVSZKY, T., AND LÉDECZI, A. Next generation (meta)modeling: web- and cloud-based collaborative tool infrastructure. In *Proceedings of the Workshop on Multi-Paradigm Modeling (MPM)* (2014), pp. 41 – 60. Cited on pages 64, 89, 90, and 93.

[192] MARÓTI, M., KERESKÉNYI, R., KECSKÉS, T., VÖLGYESI, P., AND ÁKOS LÉDECZI. Online Collaborative Environment for Designing Complex Computational Systems. *Procedia Computer Science 29*, 0 (2014), 2432 – 2441. 2014 International Conference on Computational Science. Cited on pages 43, 44, 46, 64, and 232.

[193] MCDIRMID, S. Living it up with a live programming language. In *Proceedings of OOPSLA '07* (2007), pp. 623–638. Cited on pages 186 and 187.

[194] MCDIRMID, S. Usable live programming. In *Proceedings of Onward! 2013* (2013), pp. 53–61. Cited on pages 166, 186, and 187.

[195] MELLOR, S. J., AND BALCER, M. J. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002. Cited on page 187.

[196] MEYERS, B., DESHAYES, R., LUCIO, L., SYRIANI, E., VANGHELUWE, H., AND WIMMER, M. ProMoBox: a framework for generating domain-specific property languages. In *Proceedings of the International Conference on Software Language Engineering (SLE)* (2014), pp. 1 – 20. Cited on page 5.

[197] MEYERS, B., AND VANGHELUWE, H. A framework for evolution of modelling languages. *Science of Computer Programming 76*, 12 (2011), 1223 – 1246. Cited on pages 37, 168, and 188.

[198] MOODY, D. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering 35*, 6 (2009), 756–779. Cited on pages 13 and 49.

[199] MORIN, B., BARAIS, O., JEZEQUEL, J.-M., FLEUREY, F., AND SOLBERG, A.

Models@ run.time to support dynamic adaptation. *Computer 42*, 10 (2009), 44–51. Cited on page 187.

[200] MOSTERMAN, P. J., AND VANGHELUWE, H. Computer automated multi-paradigm modeling: An introduction. *Simulation 80*, 9 (Sept. 2004), 433–450. Cited on pages 1, 3, 19, 32, and 50.

[201] MULLER, P.-A., FLEUREY, F., AND JÉZÉQUEL, J.-M. Weaving Executability into Object-oriented Meta-languages. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems* (Berlin, Heidelberg, 2005), MoDELS'05, Springer-Verlag, pp. 264–278. Cited on page 41.

[202] MULLER, P.-A., FLEUREY, F., AND JÉZÉQUEL, J.-M. Weaving executability into Object-Oriented meta-languages. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2005), pp. 264 – 278. Cited on page 93.

[203] MURATA, T. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE* (1989), pp. 541 – 580. Cited on pages 14 and 24.

[204] MUSTAFIZ, S., DENIL, J., LÚCIO, L., AND VANGHELUWE, H. The FTG+PM framework for multi-paradigm modelling: an automotive case study. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling* (2012), pp. 13–18. Cited on pages 33 and 124.

[205] MUSTAFIZ, S., AND VANGHELUWE, H. Explicit modelling of Statechart simulation environments. In *Summer Simulation Multiconference* (2013), Society for Computer Simulation International (SCS), pp. 445 – 452. Toronto, Canada. Cited on pages 128 and 215.

[206] MUZY, A., INNOCENTI, E., AIELLO, A., SANTUCCI, J.-F., AND WAINER, G. Specification of discrete event models for fire spreading. *Simulation 81*, 2 (2005), 103–117. Cited on page 114.

[207] MUZY, A., AND WAINER, G. Comparing simulation methods for fire spreading across a fuel bed. In *Proceedings of AIS'2002* (2002), pp. 219–224. Cited on page 143.

[208] NIKOUKARAN, J., HLUPIC, V., AND PAUL, R. J. Criteria for simulation software evaluation. In *Proceedings of the 1998 Winter Simulation Conference* (1998), pp. 399–406. Cited on pages xxxv and 145.

[209] NOTOWIDIGDO, M., AND MILLER, R. C. Off-line sketch interpretation. In *AAAI Fall Symposium on Making Pen-Based Interaction Intelligent and Natural* (2004), pp. 120–126. Cited on page 196.

[210] NUTARO, J. J. adevs. `http://www.ornl.gov/~1qn/adevs/`, 2015. Cited on pages 29 and 143.

[211] OAKES, B. Optimizing simulink models. Tech. Rep. CS-TR-2014.5, McGill University, 2014. Cited on page 23.

[212] ODELL, J. J. Power types. *Journal of Object-Oriented Programming 7*, 2 (1994), 8 – 12. Cited on page 39.

[213] OSLC COMMUNITY. OSLC - Open services for lifecycle collaboration core specification version 3.0. http://open-services.net, 2017. Cited on page 122.

[214] OSTERWEIL, L. Software processes are software too. In *Proceedings of the*

9th International Conference on Software Engineering (1987), ICSE '87, IEEE Computer Society Press, pp. 2–13. Cited on pages 42 and 124.

[215] OUYANG, C., VERBEEK, E., VAN DER AALST, W., BREUTEL, S., DUMAS, M., AND TER HOFSTEDE, A. H. M. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program. 67*, 2-3 (2007), 162–198. Cited on page 122.

[216] PALANIAPPAN, S., SAWHNEY, A., AND SARJOUGHIAN, H. S. Application of the DEVS Framework in Construction Simulation. In *Proceedings of the 38th Conference on Winter Simulation* (2006), Winter Simulation Conference, pp. 2077–2086. Cited on page 144.

[217] PAP, Z., MAJZIK, I., PATARICZA, A., AND SZEGI, A. Methods of checking general safety criteria in uml statechart specifications. *RELIABILITY ENGINEERING & SYSTEM SAFETY 87* (2005), 89 – 107. Cited on page 128.

[218] PAVLETIC, D., VOELTER, M., RAZA, S. A., KOLB, B., AND KEHRER, T. Extensible debugger framework for extensible languages. *Lecture Notes in Computer Science 9111* (2015), 33–49. Cited on page 164.

[219] PETRE, M. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM 38*, 6 (1995), 33–44. Cited on pages 13, 54, 69, and 186.

[220] POP, A., SJÖLUND, M., ASGHAR, A., FRITZSON, P., AND FRANCESCO, C. Static and Dynamic Debugging of Modelica Models. In *Proceedings of the 9th International Modelica Conference* (2012), pp. 443–454. Cited on page 215.

[221] POSSE, E. *Modelling and simulation of dynamic structure discrete-event systems*. PhD thesis, School of Computer Science, McGill University, Oct. 2008. Cited on page 146.

[222] POTHIER, G., AND TANTER, E. Back to the future: Omniscient debugging. *IEEE Software 26*, 6 (2009), 78–85. Cited on pages 208 and 209.

[223] POTHIER, G., TANTER, E., AND PIQUER, J. Scalable omniscient debugging. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications* (2007), OOPSLA '07, ACM, pp. 535–552. Cited on pages 209, 216, and 219.

[224] PREISS, B. R., LOUCKS, W. M., AND MACINTYRE, I. D. Effects of the checkpoint interval on time and space in time warp. *ACM Trans. Model. Comput. Simul. 4*, 3 (1994), 223–253. Cited on pages 211 and 217.

[225] QAMAR, A., HERZIG, S., AND PAREDIS, C. J. J. A domain-specific language for dependency management in model-based systems engineering. In *Proceedings of the International Workshop on Multi-Paradigm Modeling* (2013), pp. 7–16. Cited on pages 5 and 42.

[226] QUESNEL, G., DUBOZ, R., RAMAT, E., AND TRAORÉ, M. K. VLE: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Simulation Multiconference* (2007), pp. 367–374. Cited on pages 29 and 144.

[227] RABBI, F., LAMO, Y., YU, I. C., AND KRISTENSEN, L. M. A diagrammatic approach to model completion. In *Proceedings of the Workshop on the Analysis of Model Transformations (AMT)* (2015), pp. 56 – 65. Cited on page 37.

[228] RABBI, F., LAMO, Y., YU, I. C., AND KRISTENSEN, L. M. WebDPF: A web-based

metamodelling and model transformation environment. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development* (2016), pp. 87–98. Cited on pages 43 and 89.

[229] REGGIO, G., LEOTTA, M., AND RICCA, F. Who knows/uses what of the uml: A personal opinion survey. *Lecture Notes in Computer Science 8767* (2014), 149–165. Cited on page 130.

[230] REINHARTZ-BERGER, I., STURM, A., AND CLARK, T. Exploring multi-level modeling relations using variability mechanisms. In *Proceedings of the Workshop on Multi-Level Modelling (MULTI)* (2015), pp. 23 – 32. Cited on pages 40 and 90.

[231] REITER, T., RETSCHITZEGGER, W., AND ALTMANNINGER, K. Think global, act local: Implementing model management with domain-specific integration languages. In *Proceedings of the International Workshop on MPM* (2006), pp. 51–66. Cited on page 42.

[232] RENSINK, A. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)* (2004), Lecture Notes in Computer Science, pp. 479–485. Cited on page 44.

[233] ROHR, M., BOSKOVIC, M., GIESECKE, S., AND HASSELBRING, W. Model-driven development of self-managing software systems. In *Proceedings of the Models at run.time workshop co-located with the ACM/IEEE 9th International Conference MODELS 2006* (2006). Cited on page 187.

[234] ROLLAND, C. A comprehensive view of process engineering. In *Proceedings of the International Conference on Advanced Information Systems Engineering* (1998), pp. 1–24. Cited on page 42.

[235] ROMERO, J. R., RIVERA, J. E., DURÁN, F., AND VALLECILLO, A. Formal and tool support for Model Driven Engineering with Maude. *Journal of Object Technology 6*, 9 (2007), 187 – 207. Cited on pages 36, 90, and 93.

[236] RÖNNGREN, R., AND AYANI, R. Adaptive checkpointing in time warp. *SIGSIM Simul. Dig. 24*, 1 (1994), 110–117. Cited on pages 211 and 217.

[237] ROSE, L., GUERRA, E., DE LARA, J., ETIEN, A., KOLOVOS, A., AND PAIGE, R. Genericity for model management operations. *Software and Systems Modeling (SoSyM) 12*, 1 (2013), 201 – 219. Cited on pages 5, 90, and 109.

[238] ROSE, L. M., PAIGE, R. F., KOLOVOS, D. S., AND POLACK, F. A. C. The Epsilon Generation Language. *Lecture Notes in Computer Science 5095* (2008), 1–16. Cited on page 41.

[239] ROSSINI, A., DE LARA, J., GUERRA, E., RUTLE, A., AND WOLTER, U. A formalisation of deep metamodelling. *Formal Aspects of Computing 26*, 6 (2014), 1115–1152. Cited on page 90.

[240] ROUSSEAU, A., HALBACH, S., MICHAELS, L., SHIDORE, N., KIM, N., KIM, N., KARBOWSKI, D., AND KROPINSKI, M. Electric drive vehicle development and evaluation using system simulation. In *Proceedings of the 19th IFAC World Congress* (2014), pp. 7886–7891. Cited on page 2.

[241] RUSSELL, N., VAN DER AALST, W., TER HOFSTEDE, A., AND WOHED, P. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Proceedings of the Asia-Pacific Conference on Conceptual Modelling* (2006), pp. 95–104. Cited on page 42.

[242] SAFA, L. The practice of deploying DSM report from a japanese appliance maker trenches. In *Proceedings of the OOPSLA Workshop on Domain-Specific Modeling* (2006), pp. 185–196. Cited on pages 2 and 4.

[243] SALAY, R., CHECHICK, M., EASTERBROOK, S., DISKIN, Z., MCCORMICK, P., NEJATI, S., SABETZADEH, M., AND VIRIYAKATTIYAPORN, P. An Eclipse-based tool framework for software model management. In *Proceedings of the Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2007), pp. 55 – 59. Cited on pages 42, 45, and 90.

[244] SALAY, R., AND CHECHIK, M. Supporting agility in MDE through modeling language relaxation. In *Proceedings of the Workshop on Extreme Modeling* (2013), pp. 21 – 30. Cited on page 37.

[245] SALAY, R., AND CHECHIK, M. Supporting agility in MDE through modeling language relaxation. In *Proceedings of the Workshop on Extreme Modeling co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2013).* (2013), pp. 20–27. Cited on page 97.

[246] SALAY, R., KOKALY, S., DI SANDRO, A., AND CHECHIK, M. Enriching megamodel management with collection-based operators. In *Proceedings of MoDELS 2015* (2015), pp. 236–245. Cited on pages 42 and 109.

[247] SALAY, R., MYLOPOULOS, J., AND EASTERBROOK, S. Managing models through macromodeling. In *Proceedings of the International Conference on Automated Software Engineering (ASE)* (2008), pp. 447 – 450. Cited on page 100.

[248] SALAY, R., ZSCHALER, S., AND CHECHIK, M. Transformation reuse: What is the intent? In *Proceedings of AMT@MoDELS* (2015), pp. 1–7. Cited on page 40.

[249] SANDEWALL, E. Programming in an interactive environment: The "lisp" experience. *ACM Comput. Surv. 10*, 1 (1978), 35–71. Cited on page 186.

[250] SARJOUGHIAN, H., AND ZEIGLER, B. DEVSJava: Basis for a DEVS-based Collaborative M&S Environment. *SIMULATION 30* (1998), 29–36. Cited on page 143.

[251] SARJOUGHIAN, H. S., AND CHEN, Y. Standardizing DEVS models: an endogenous standpoint. In *Proceedings of the 2011 Spring Simulation Multiconference* (2011), pp. 266–273. Cited on page 29.

[252] SCHIFFELERS, R., ALBERTS, W., AND VOETEN, J. Model-based specification, analysis and synthesis of servo controllers for lithoscanners. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling (MPM 2012)* (2012), pp. 55–60. Cited on page 2.

[253] SCHIFFELERS, R. R., THEUNISSEN, R. J., VAN BEEK, D. A., AND ROODA, J. E. Model-based engineering of supervisory controllers using CIF. *ECEASST 21* (2009). Cited on page 2.

[254] SCHMIDT, K. LoLA: A low level analyser. *Lecture Notes in Computer Science 1825* (2000), 465–474. Cited on page 54.

[255] SCHUSTER, A., AND SPRINKLE, J. Synthesizing executable simulations from structural models of component-based systems. *ECEASST* (2009). Cited on page 3.

[256] SEN, S., BAUDRY, B., AND VANGHELUWE, H. Towards domain-specific model editors with automatic model completion. *Simulation 3*, 12 (2010), 109 – 126. Cited

on pages 35, 36, and 37.

[257] SENDALL, S., AND KOZACZYNSKI, W. Model transformation: The heart and soul of model-driven software development. *IEEE Softw. 20*, 5 (2003), 42–45. Cited on pages 3, 15, and 130.

[258] SEO, C., ZEIGLER, B. P., COOP, R., AND KIM, D. DEVS modeling and simulation methodology with MS4Me software. In *Proceedings of the 2013 Spring Simulation Multiconference* (2013), pp. 33:1–33:8. Cited on pages 29 and 144.

[259] SHANG, H., AND WAINER, G. A model of virus spreading using Cell-DEVS. In *Computational Science ICCS 2005*, vol. 3515 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 145–201. Cited on page 143.

[260] SILVER, B., AND RICHARD, B. *BPMN method and style*, vol. 2. Cody-Cassidy Press Aptos, 2009. Cited on page 122.

[261] SIX, J. M., AND TOLLIS, I. G. Circular drawings of biconnected graphs. In *Algorithm Engineering and Experimentation* (1999), pp. 57–73. Cited on page 191.

[262] SLOANE, A., ROBERTS, M., BUCKLEY, S., AND MUSCAT, S. Monto: A disintegrated development environment. In *Proceedings of the International Conference on Software Language Engineering* (2014), pp. 211–220. Cited on pages 189 and 205.

[263] SORENSEN, A., AND GARDNER, H. Programming with time: cyber-physical programming with Impromptu. In *Proceedings of Onward! 2010* (2010), pp. 822–834. Cited on page 187.

[264] SOTTET, J.-S., AND BIRI, N. JSMF: a Javascript flexible modelling framework. In *Proceedings of the 2nd Workshop on Flexible Model Driven Engineering* (2016), pp. 42–51. Cited on page 97.

[265] SOUSA, V., AND SYRIANI, E. An expeditious approach to modeling ide interaction design. In *Joint Proceedings of the 3rd International Workshop on the Globalization Of Modeling Languages and the 9th International Workshop on Multi-Paradigm Modeling* (2015), pp. 52–61. Cited on page 194.

[266] STEEL, J., AND JÉZÉQUEL, J.-M. Typing relationships in MDA. In *Proceedings of the European Workshop on Model Driven Architecture (MDA)* (2004), pp. 154 – 159. Cited on pages 35, 37, and 89.

[267] STEEL, J., AND JÉZÉQUEL, J.-M. Model typing for improving reuse in Model-Driven Engineering. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2005), pp. 84 – 96. Cited on pages 35, 36, and 89.

[268] STEWART, D., AND CHAKRAVARTY, M. M. Dynamic applications from the ground up. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell* (2005), pp. 27–38. Cited on page 187.

[269] STIEHL, V. *Process-Driven Applications with BPMN*. Springer, 2014. Cited on page 121.

[270] STOLLON, N. *On-Chip Instrumentation*, 1st ed. Springer, 2011. Cited on page 219.

[271] SYRIANI, E., AND VANGHELUWE, H. A modular timed graph transformation language for simulation-based design. *Software and Systems Modeling (SoSyM) 12*, 2 (2013), 387–414. Cited on page 15.

[272] SYRIANI, E., VANGHELUWE, H., AND AL MALLAH, A. Modelling and simulation-

based design of a distributed DEVS simulator. In *Proceedings of the Winter Simulation Conference* (2011), pp. 3007–3021. Cited on page 153.

[273] SYRIANI, E., VANGHELUWE, H., MANNADIAR, R., HANSEN, C., VAN MIERLO, S., AND ERGIN, H. AToMPM: a web-based modeling environment. In *Joint Proceedings of MoDELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition* (2013), pp. 21 – 25. Cited on pages 43, 44, 89, 90, 91, 100, 109, 203, and 208.

[274] SZÁRNYAS, G., IZSÓ, B., RÁTH, I., HARMATH, D., BERGMANN, G., AND VARRÓ, D. IncQuery-D: A distributed incremental model query framework in the cloud. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2014), pp. 653 – 669. Cited on pages 41, 101, and 202.

[275] SZTIPANOVITS, J., KARSAI, G., AND FRANKE, H. Model-integrated program synthesis environment. In *IEEE Symposium on Engineering of Computer Based Systems* (1996). Cited on page 3.

[276] TANIMOTO, S. L. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing 1* (1990), 127–139. Cited on pages 164, 186, and 187.

[277] TAROMIRAD, M., MATRAGKAS, N., AND PAIGE, R. F. Towards a multi-domain model-driven traceability approach. In *Proceedings of the International Workshop on MPM* (2013), pp. 27–36. Cited on pages 4 and 5.

[278] TEWOLDEBERHAN, T. W., VERBRAECK, A., VALENTIN, E., AND BARDONNET, G. An evaluation and selection methodology for discrete-event simulation software. In *Proceedings of the 2002 Winter Simulation Conference* (Dec. 2002), pp. 67–75. Cited on pages xxxv and 145.

[279] THEISZ, Z., AND MEZEI, G. An algebraic instantiation technique illustrated by multilevel design patterns. In *Proceedings of the Workshop on Multi-Level Modelling (MULTI)* (2015), pp. 53 – 62. Cited on pages 35, 89, 90, 93, and 97.

[280] TISI, M., DOUENCE, R., AND WAGELAAR, D. Lazy evaluation for OCL. In *Proceedings of the International Workshop on OCL and Textual Modeling* (2015), pp. 46–61. Cited on page 41.

[281] TOLVANEN, J.-P., AND KELLY, S. Defining domain-specific modeling languages to automate product derivation: Collected experiences. *Lecture Notes in Computer Science 3714* (2005), 198 – 209. Cited on page 3.

[282] UJHELYI, Z., BERGMANN, G., HEGEDÜS, A., HORVÁTH, A., IZSÓ, B., RÁTH, I., SZATMÁRI, Z., AND VARRÓ, D. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming 98*, 1 (2015), 80–99. Cited on pages 41 and 202.

[283] UNGAR, D., AND SMITH, R. B. SELF: the power of simplicity. In *Proceedings of the Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (1987), pp. 227 – 242. Cited on pages 92 and 186.

[284] VAN DER AALST, W. Business process management as the "killer app" for petri nets. *Software and System Modeling 14*, 2 (2015), 685–691. Cited on page 121.

[285] VAN DER AALST, W., AND TER HOFSTEDE, A. YAWL: yet another workflow language. *Inf. Syst. 30*, 4 (2005), 245–275. Cited on page 122.

[286] VAN DER AALST, W., TER HOFSTEDE, A., KIEPUSZEWSKI, B., AND BARROS, A. Workflow patterns. *Distributed and Parallel Databases 14*, 1 (2003), 5–51. Cited on page 121.

[287] VAN DER AALST, W. M. P., VAN HEE, K. M., TER HOFSTEDE, A. H. M., SIDOROVA, N., VERBEEK, H. M. W., VOORHOEVE, M., AND WYNN, M. T. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing 23*, 3 (2011), 333–363. Cited on page 130.

[288] VAN DER CRUYSSE, J. Just-in-time compiler for the modelverse, 2017. Cited on page 226.

[289] VAN DER STORM, T. Semantic deltas for live DSL environments. In *Proceedings of the 1st International Workshop on Live Programming* (Piscataway, NJ, USA, 2013), LIVE '13, IEEE Press, pp. 35–38. Cited on pages 164, 165, and 187.

[290] VAN DER STRAETEN, R. Towards a methodology for semantics specification of domain-specific models through properties. *ECEASST* (2011). Cited on page 4.

[291] VAN GORP, P., SCHIPPERS, H., DEMEYER, S., AND JANSSENS, D. Students can get excited about formal methods: a model-driven course on petri-nets, metamodels and graph grammars. In *MoDELS Educators' Symposium* (2007), pp. 1–10. Cited on page 3.

[292] VAN GORP, P., SCHIPPERS, H., DEMEYER, S., AND JANSSENS, D. Transformation techniques can make students excited about formal methods. *Information & Software Technology 50*, 12 (2008), 1295–1304. Cited on page 3.

[293] VAN MIERLO, S. Explicitly modelling model debugging environments. In *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015* (2015), pp. 24–29. Cited on pages 132, 187, and 208.

[294] VAN MIERLO, S. *A Multi-Paradigm Modelling Approach for Engineering Model Debugging Environments*. PhD thesis, University of Antwerp, 2018. Cited on pages 164 and 208.

[295] VAN MIERLO, S., BARROCA, B., VANGHELUWE, H., SYRIANI, E., AND KÜHNE, T. Multi-level modelling in the modelverse. In *Proceedings of the Workshop on Multi-Level Modelling (MULTI)* (2014), pp. 83 – 92. Cited on pages 38, 90, and 104.

[296] VAN MIERLO, S., VAN TENDELOO, Y., BARROCA, B., MUSTAFIZ, S., AND VANGHELUWE, H. Explicit modelling of a Parallel DEVS experimentation environment. In *Proceedings of the 2015 Spring Simulation Multiconference* (2015), SpringSim '15, Society for Computer Simulation International, pp. 860–867. Cited on pages 144 and 208.

[297] VAN MIERLO, S., VAN TENDELOO, Y., DÁVID, I., MEYERS, B., GEBREMICHAEL, A., AND VANGHELUWE, H. A multi-paradigm approach for modelling service interactions in model-driven engineering processes. In *Proceedings of Mod4Sim* (2018), Mod4Sim, part of the Spring Simulation Multi-Conference, pp. 565–576. Cited on page 155.

[298] VAN MIERLO, S., VAN TENDELOO, Y., MEYERS, B., EXELMANS, J., AND VANGHELUWE, H. SCCD: SCXML extended with class diagrams. In *Proceedings of the Workshop on Engineering Interactive Systems with SCXML* (2016), pp. 2:1–2:6. Cited on pages 26, 69, 70, 119, 122, 148, and 202.

[299] VAN MIERLO, S., VAN TENDELOO, Y., MEYERS, B., AND VANGHELUWE, H. *Domain-Specific Modelling for Human-Computer Interaction*. Springer, 2017, pp. 435 – 463. Cited on page 189.

[300] VAN MIERLO, S., VAN TENDELOO, Y., AND VANGHELUWE, H. Debugging Parallel DEVS. *SIMULATION 93*, 4 (2017), 285–306. Cited on pages 187, 208, and 219.

[301] VAN MIERLO, S., AND VANGHELUWE, H. Adding rule-based model transformation to modelling languages in MetaEdit+. *ECEAST 54* (2012). Cited on page 45.

[302] VAN ROZEN, R., AND VAN DER STORM, T. Towards live domain-specific languages: from text differencing to adapting models at run time. *Software & Systems Modeling* (2017), 1–18. Cited on pages 164 and 187.

[303] VAN TENDELOO, Y. Activity-aware DEVS simulation. Master's thesis, University of Antwerp, Antwerp, Belgium, 2014. Cited on pages 109 and 219.

[304] VAN TENDELOO, Y. Foundations of a multi-paradigm modelling tool. In *MoDELS ACM Student Research Competition* (2015), pp. 52–57. Cited on pages 121, 155, and 182.

[305] VAN TENDELOO, Y., BARROCA, B., VAN MIERLO, S., AND VANGHELUWE, H. Modelverse specification. Tech. rep., University of Antwerp, 2017. Cited on page 63.

[306] VAN TENDELOO, Y., VAN MIERLO, S., AND VANGHELUWE, H. Time- and space-conscious omniscient debugging of parallel DEVS. In *Proceedings of the 2017 Symposium on Theory of Modeling and Simulation - DEVS* (Apr. 2017), TMS/DEVS '17, part of the Spring Simulation Multi-Conference, Society for Computer Simulation International, pp. 1001 – 1012. Cited on pages 212 and 219.

[307] VAN TENDELOO, Y., AND VANGHELUWE, H. Activity in PythonPDEVS. In *Proceedings of ACTIMS 2014* (2014). Cited on page 109.

[308] VAN TENDELOO, Y., AND VANGHELUWE, H. The modular architecture of the Python(P)DEVS simulation kernel. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS* (2014), pp. 387–392. Cited on pages 121, 143, 144, 208, and 215.

[309] VAN TENDELOO, Y., AND VANGHELUWE, H. PythonPDEVS: a distributed Parallel DEVS simulator. In *Proceedings of the 2015 Spring Simulation Multiconference* (2015), SpringSim '15, Society for Computer Simulation International, pp. 844–851. Cited on pages 143 and 219.

[310] VAN TENDELOO, Y., AND VANGHELUWE, H. An overview of PythonPDEVS. In *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications* (Apr. 2016), C. W. RED, Ed., Éditions Cépaduès, pp. 59 – 66. Cited on page 143.

[311] VAN TENDELOO, Y., AND VANGHELUWE, H. Teaching the fundamentals of the modelling of cyber-physical systems. In *Proceedings of the 2016 Symposium on Theory of Modeling and Simulation - DEVS* (Apr. 2016), TMS/DEVS '16, part of the Spring Simulation Multi-Conference, Society for Computer Simulation International, pp. 646 – 653. Cited on page 3.

[312] VAN TENDELOO, Y., AND VANGHELUWE, H. Classic DEVS modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference* (Dec. 2017), WSC 2017, IEEE, pp. 644 – 656. Cited on page 27.

[313] VAN TENDELOO, Y., AND VANGHELUWE, H. An evaluation of DEVS simulation tools. *SIMULATION 93*, 2 (2017), 103–121. Cited on pages xxxv, 143, 145, and 146.

[314] VAN TENDELOO, Y., AND VANGHELUWE, H. Explicitly modelling the type/instance relation. In *Proceedings of MODELS 2017 Satellite Event* (Sept. 2017), Ceur-WS, pp. 393 – 398. Cited on pages 102, 108, and 154.

[315] VAN TENDELOO, Y., AND VANGHELUWE, H. Increasing performance of a DEVS simulator by means of computational resource usage "activity" models. *SIMULATION 93*, 12 (2017), 1045 – 1061. Cited on page 109.

[316] VAN TENDELOO, Y., AND VANGHELUWE, H. The Modelverse: a tool for multi-paradigm modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference* (Dec. 2017), WSC 2017, IEEE, pp. 944 – 955. Cited on pages 3, 50, 121, 147, and 182.

[317] VAN TENDELOO, Y., AND VANGHELUWE, H. Extending the DEVS formalism with initialization information. *ArXiv e-prints* (2018). Cited on pages xxxv, 29, 145, and 146.

[318] VAN TENDELOO, Y., AND VANGHELUWE, H. Unifying model- and screen sharing. In *Proceedings of the 2018 WETICE conference* (June 2018), IEEE, pp. 127–132. Cited on page 205.

[319] VANGHELUWE, H. DEVS as a common denominator for multi-formalism hybrid systems modelling. *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design (CACSD)* (2000), 129–134. Cited on page 109.

[320] VANGHELUWE, H., RIEGELHAUPT, D., MUSTAFIZ, S., DENIL, J., AND VAN MIERLO, S. Explicit modelling of a CBD experimentation environment. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS* (2014), TMS/DEVS '14, part of the Spring Simulation Multi-Conference, Society for Computer Simulation International, pp. 379–386. Cited on page 187.

[321] VANGHELUWE, H. L., VANSTEENKISTE, G. C., AND KERCKHOFFS, E. J. Simulation for the future: Progress of the Esprit basic research working group 8467. In *Proceedings of the European Simulation Symposium (ESS)* (1996), pp. XXIX – XXXIV. Cited on page 3.

[322] VANHERPEN, K., DENIL, J., DÁVID, I., DE MEULENAERE, P., MOSTERMAN, P. J., TÖRNGREN, M., QAMAR, A., AND VANGHELUWE, H. Ontological reasoning for consistency in the design for Cyber-Physical Systems. In *Proceedings of the Workshop on Cyber-Physical Production Systems (CPPS)* (2016), pp. 9:1 – 9:8. Cited on page 38.

[323] VARRÓ, D., AND PATARICZA, A. Generic and meta-transformations for model transformation engineering. In *Proceedings of the Conference on the Unified Modeling Language (UML)* (2004), pp. 290 – 304. Cited on pages 90 and 98.

[324] VARRÓ, G., FRIEDL, K., AND VARRÓ, D. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electronic Notes in Theoretical Computer Science 152* (2006), 191–205. Cited on page 41.

[325] VIGNAGA, A., JOUAULT, F., BASTARRICA, M. C., AND BRUNELIÈRE, H. Typing in model management. *Lecture Notes in Computer Science 5563* (2009). Cited on page 109.

[326] VOELTER, M., SIEGMUND, J., BERGER, T., AND KOLB, B. Towards user-friendly

projectional editors. In *Proceedings of the International Conference on Software Language Engineering* (2014), pp. 41–61. Cited on page 205.

[327] VON DETTEN, M., HEINZEMANN, C., PLATENIUS, M. C., RIEKE, J., TRAVKIN, D., AND HILDEBRANDT, S. Story diagrams-syntax and semantics. Tech. Rep. tr-ri-12-324, University of Paderborn, 2012. Cited on page 122.

[328] WAINER, G. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience 32*, 13 (2002), 1261–1306. Cited on pages 29 and 143.

[329] WAINER, G., AND GIAMBIASI, N. Application of the Cell-DEVS Paradigm for Cell Spaces Modelling and Simulation. *SIMULATION 76*, 1 (2001), 22–39. Cited on page 143.

[330] WEERAWARANA, S., CURBERA, F., LEYMANN, F., STOREY, T., AND FERGUSON, D. F. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005. Cited on page 122.

[331] WEGNER, P. Concepts and paradigms of Object-Oriented programming. *SIGPLAN Object-Oriented Programming Systems (OOPS) Messenger 1*, 1 (1990), 7 – 87. Cited on page 93.

[332] WIMMER, M., KUSEL, A., RETSCHITZEGGER, W., SCHÖNBÖCK, J., SCHWINGER, W., CUADRADO, J. S., GUERRA, E., AND DE LARA, J. Reusing model transformations across heterogeneous metamodels. *ECEASST* (2011). Cited on page 41.

[333] WU, H., GRAY, J., AND MERNIK, M. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience 38*, 10 (2008), 1073–1103. Cited on page 164.

[334] XIA, Y., LIU, Y., LIU, J., AND ZHU, Q. Modeling and performance evaluation of BPEL processes: A stochastic-petri-net-based approach. *IEEE Trans. Systems, Man, and Cybernetics, Part A 42*, 2 (2012), 503–510. Cited on page 122.

[335] ZAYTSEV, V., AND BAGGE, A. H. Parsing in a broad sense. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)* (2014), pp. 50 – 67. Cited on page 205.

[336] ZEIGLER, B., SEO, C., AND KIM, D. System entity structures for suites of simulation models. *International Journal of Modeling, Simulation, and Scientific Computing 4* (2013), 3:1–3:11. Cited on page 144.

[337] ZEIGLER, B. P. *Multi-faceted Modelling and Discrete-Event Simulation*. Academic Press, 1984. Cited on pages xxx and 48.

[338] ZEIGLER, B. P., PRAEHOFER, H., AND KIM, T. G. *Theory of Modeling and Simulation*, second ed. Academic Press, 2000. Cited on pages 14, 27, 120, and 143.

[339] ZEIGLER, B. P., SEO, C., COOP, R., AND KIM, D. Creating Suites of Models with System Entity Structure: Global Warming Example. In *Proceedings of the 2013 Spring Simulation Multiconference* (2013), pp. 32:1–32:8. Cited on page 144.

[340] ZELKOWITZ, M. V. Reversible execution. *Communications of the ACM 16*, 9 (1973), 566–566. Cited on page 216.

[341] ZELLAG, K., AND VANGHELUWE, H. Modelling- and simulation-based design of multi-tier systems. *ECEASST* (2011). Cited on page 3.

[342] ZELLER, A. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. Cited on page 164.

[343] ZHANG, Y., AND XU, B. A survey of semantic description frameworks for programming languages. *SIGPLAN Notices 39*, 3 (2004), 14–30. Cited on page 15.

[344] ZHAO, Q., RABBAH, R., AMARASINGHE, S., RUDOLPH, L., AND WONG, W.-F. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. *Lecture Notes in Computer Science 4959* (2008). Cited on page 219.