UNIVERSITEIT ANTWERPEN

A Multi-Paradigm Modelling Approach for Engineering Model Debugging Environments

Een Multi-Paradigma Modelleeraanpak voor het Ontwikkelen van Debugomgevingen voor Modelbouw

Auteur:
Simon VAN MIERLO

Promotor:
Prof. Dr. Hans VANGHELUWE

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

Contents

Acknowledgements

I would like to take this opportunity to thank a number of people that have played an important role supporting me during my 4-year PhD project.

First off, this thesis would not be here, and I probably would not have started a scientific career, if it weren't for my supervisor, Hans Vangheluwe. You created numerous opportunities for me to discover the scientific community and work within it. You taught me the scientific method, always with encouragements, rather than dictating. Thank you for instilling part of the passion you have for scientific research and being an inspiration to finish this work.

Thanks to my colleagues, not only for the scientific discussions, but equally for the team spirit, off-topic discussions, and fun activities. In particular, thank you Yentl, for sharing an office with me (and putting up with me) for all these years. Without the cross-fertilization and collaborations that emerged, this thesis would not be what it is today. Thank you Claudio, Istvan, Ken, Joachim, Bart, as well as the other members of the Ansymo research group, for being great colleagues. To Bentley, Maris, Sadaf, and Levi: thank you for hosting me on numerous occasions when I visited MSDL at McGill. To my extended colleagues in the department, the faculty, and the administration: thank you for you helpfulness, your flexibility at times, and the pleasant co-operations.

Thanks to everyone at Autodesk Research for an interesting three-month internship. In particular, thank you Rhys, Simon, and Azam for the fruitful collaboration. Due to your insights and by giving me a glimpse into how your department conducts research, my own research skills have vastly improved. Thank you Peter, Louise, Marcus, Gabrielle, and Harrison for making me feel at home in Toronto. I hope we meet again soon.

Every PhD thesis has to be reviewed and accepted by the scientific community. Thank you to the members of my PhD committee and jury: Jeff Gray, Clark Verbrugge, Tom Mens, Dirk Janssens, Jan Broeckhove, and Serge Demeyer. Your diligent reading of my thesis has improved it tremendously, and the questions I received made me rethink and adjust, arriving at this final version.

Thank you mom, for always being there for me and supporting me at every step in my life. Your continuous wisdom, care, and love have given me the strength to continue and find myself at times. Thank you dad, for your continuous support and a listening ear throughout the years.

A big thank you to my friends, who offered the necessary distractions outside of the university. Michiel, Jo, Laurens: thank you for all the years we've been friends, our longlasting friendship is truly important to me. Joachim, Michiel, Cis, and Philippe: I had a fun time studying with you at the university, and whenever we see each other (usually over a beer). Thank you to everyone I met, worked with, and had a lot of fun with in the scouts, and in EESTEC.

And last, but certainly not least, thank you Stien. Thank you for always supporting me, being there for me, and understanding how important my work is for me. Marrying you was the best decision that I made, and I cannot wait to spend the rest of my life with you.

Simon Van Mierlo
12 March 2018

Abstract

Today's engineered systems are becoming increasingly complex. This complexity stems from the demand on their autonomy, safety, and intercommunication: software-intensive systems combine a mechanical aspect with a software controller, forming a feedback loop; cyber-physical systems combine multiple independent systems into a network to achieve a higher-level task. Safety needs to be guaranteed, as our society increasingly depends on these systems to fulfil day-to-day needs. To successfully develop such systems, the methods to design, analyse and deploy such systems need to evolve. Before, systems were categorized either as a mechanical system or as a software system, and specialized development methods existed for each category. This no longer suffices: the tight integration between physical aspects and software aspects of systems needs to be acknowledged during all phases of the development cycle: from requirements engineering, to system design and validation, and ultimately deployment.

Model-Driven Engineering (MDE) is a method for developing systems that regards models as first-class citizens in the development process. When developing systems according to the MDE process, engineers build models (abstractions) that describe the different parts of the system in (graphical) modelling languages. The modelling languages provide abstractions that allow engineers to focus on *what* the system is supposed to do, instead of *how* the system is supposed to do it (which are implementation details that are important for deployment, not at design time). Within MDE, *Multi-Paradigm Modelling* (MPM) advocates to model every (relevant) aspect of a system explicitly, at the most appropriate level(s) of abstraction, using the most appropriate modelling language(s), while modelling the development process(es) explicitly. This allows specialists in a particular domain to model systems using abstractions they are most familiar with. But, while modelling languages are tools that allow domain experts to more intuitively develop their models and thereby manage the development process for highly complex systems, they do not guarantee the developed models are without defects. The complexity of the systems does not decrease by using a different development method: while the languages are more intuitive and specialized, the systems they can model are not simple by any means. By identifying system properties that are not satisfied (signifying that a requirement was not implemented by the system), early verification and validation techniques are able to discover and observe the *failures* (or *bugs*) of a system. Yet, once a failure has been observed, it is then necessary to identify why the failure occurs (*i.e.*, the defect that caused the failure), and how to modify the models to remove the cause of the failure. Locating and fixing the cause of a failure can be accomplished manually given a good understanding of the models. Alternatively, a wide range of *debugging techniques* can be used to assist developers in finding the cause of the problem. Such debugging techniques have been extensively researched and developed for systems implemented with (imperative) program code. Yet, when it comes to models, very few debugging tools and techniques are available.

Building tools for the large set of (semantically varying) modelling languages is a costly endeavour, as the users demand their tools to solve increasingly difficult tasks, such as automatic verification, symbolic execution, and design space exploration. For often-used (general-purpose) languages, this investment can be made, but for domain-specific languages, the cost of developing tools needs to be low in order for the benefits to outweigh these costs. Recent advances in modelling language engineering have provided language engineers with tools to quickly develop a specialized modelling language and its associated tools, including (graphical) modelling environments, simulators, and code generators. Language engineering techniques are used by language engineers that are tasked with developing a new language and its tooling. More recently, techniques focus on generating not only a design environment for the language, but also a runtime language for visualizing the state of the system after and during simulation, an output language for viewing and replaying a simulation trace, and a property language for defining temporal properties the system needs to satisfy.

Finding a systematic way of implementing tools and techniques (*i.e.*, finding an engineering process) for model debugging is related to the diverse set of models and modelling languages used in a typical system development. Developing such debuggers is complicated because of the interaction between the modelling language's semantics (which can exhibit, amongst others, non-deterministic, concurrent, timed, and event-processing behaviour), the different notions of simulated time (real-time versus as-fast-as-possible) and the user interacting with the execution of the model through a (visual) debugging interface. We contribute to the state of the art in language engineering by providing a structured approach for turning modelling and simulation environments into interactive debugging environments. To achieve this, we present:

- an approach to instrument simulators with debugging support based on an explicit representation of their control flow;
- an architecture, that leverages existing components for debugging purposes, including the (graphical) modelling environment, the debugging-enhanced simulator, and any model-specific visualizations;
- a workflow that guides language engineers and tool builders to enhance their languages and tools with debugging support.

To demonstrate feasibility and validate our approach, we apply it to a set of formalisms with diverse semantics. These formalisms are carefully chosen based on an analysis of their semantic properties. For each language, we define a set of useful debugging operations and implement them by instrumenting their simulation algorithm using our technique based on an explicit model of their control flow. We focus on interactive debugging techniques which allow users to control and observe the simulation process. The user controls the debugger through a (graphical) interface which communicates with the debugging-enhanced simulator. These interfaces use the abstractions provided by the debugged language to present intermediate states of the simulation to the user. The interfaces are generated from a definition of a *debugging language*, which extends the *runtime language* of the modelling language. We also explain how two advanced debugging techniques that were implemented for a number of programming languages can be (efficiently) implemented for modelling languages: omniscient debugging (which allows users to step back in a simulation trace) and live modelling (which allows users to modify the design model during simulation). For these advanced techniques, we provide a generic workflow and techniques for implementing these operations for any modelling language, and apply these techniques to representative examples to demonstrate feasibility.

Nederlandstalige Samenvatting

De complexiteit van systemen die door de mens worden ontwikkeld stijgt continu. Dit komt doordat er hoge eisen worden gesteld aan de veiligheid van deze systemen, hun autonomie, en hun intercommunicatie: software-intensieve systemen combineren een mechanisch aspect met software die deze controleert (waardoor een terugkoppeling ontstaat); cyber-fysische systemen plaatsen verschillende onafhankelijke systemen in een netwerk om zo een taak te verrichten. Om deze systemen succesvol te kunnen ontwikkelen, moeten de methodes om deze systemen te ontwerpen, analyseren, en uitrollen mee evolueren. Vroeger werden systemen ingedeeld als een mechanisch of een softwaresysteem, en er bestonden gespecialiseerde ontwikkelmethoden voor elk type systemen. Dit voldoet niet langer: de integratie tussen de fysieke en software aspecten van systemen moet in beschouwing worden genomen tijdens alle fasen van systeemontwikkeling: van het verzamelen van benodigdheden, tot het ontwerp en validatie van het systeem, en uiteindelijk het uitrollen van het systeem.

Model-Driven Engineering (MDE) is een methode voor het ontwikkelen van systemen, waarbij modellen die het gedrag en de structuur van systemen beschrijven centraal staan. Ingenieurs bouwen modellen (abstracties) van de verschillende delen van het systeem in (grafische) modelleertalen. Deze talen laten de ingenieurs toe om te focussen op wat het systeem moet doen, in plaats van hoe het systeem zijn taken moet volbrengen. Binnen MDE beschrijft *Multi-Paradigm Modelling* (MPM) een methode waarbij alle relevante aspecten van systemen expliciet gemodelleerd worden, op het (de) meest geschikte niveau(s) van abstractie, gebruik makende van de meest geschikte modelleertaal (of -talen), waarbij de ontwikkelprocessen expliciet gemodelleerd worden. Dit laat domeinexperts toe om hun systemen te ontwikkelen met notaties waar zij vertrouwd mee zijn. Maar, hoewel dit het ontwikkelproces vergemakkelijkt, is er geen garantie dat het ontwerp van het systeem geen fouten bevat. De complexiteit van systemen verlaagt niet door een andere ontwikkelmethode te gebruiken: hoewel de ontwerptalen meer gespecialiseerd en intuïtief zijn, zijn de systemen die ermee gemodelleerd worden allesbehalve triviaal. Door de eigenschappen te identificeren die werden gespecifieerd voor het systeem, maar waaraan het geïmplementeerde systeem niet voldoet, kunnen verificatie en validatie technieken de fouten (of *bugs*) van het systeem ontdekken en observeren. Het is noodzakelijk, eens een fout geobserveerd wordt, de oorzaak hiervan (een *defect*) te ontdekken. De defecten kunnen gevonden worden door manuele inspectie van de modellen, indien de modellen goed begrepen worden. Een alternatief voor manuele inspectie, die ontwikkelaars bijstaat bij het vinden van defecten, wordt geboden door *debug*technieken. In het domein van programmeertalen zijn deze technieken uitvoerig onderzocht, maar voor MDE en MPM zijn ze vaak onbestaande, of geïmplementeerd op een ad-hoc manier.

Het ontwikkelen en onderhouden van *tools* voor een grote verzameling (semantisch verschillende) modelleertalen is een grote investering, zeker gezien de gebruikers van deze tools verwachten dat meer en meer complexe taken worden uitgevoerd, zoals automatische verificatie, symbolische uitvoering, en het verkennen van de ontwerpruimte. Voor vaak gebruikte modelleertalen kan deze investering gemaakt worden, maar voor domein-specifieke talen moet de ontwikkelkost gedrukt worden. Recente vooruitgang in het ontwerpen van modelleertalen heeft gezorgd voor tools en technieken die taalingenieurs in staat stellen om snel een gespecialiseerde modelleertaal en bijhorende tools te ontwikkelen, waaronder (visuele) modelleeromgevingen, simulatoren, en codegeneratoren. Meer recent werden technieken ontwikkeld om niet enkel een ontwerptaal voor een modelleertaal te construeren, maar ook een *runtime* taal, die de staat van het systeem kan visualiseren tijdens en na simulatie, een output taal om een simulatie *trace* te bekijken en terug af te spelen, en een taal waarin temporele eigenschappen waaraan het systeem moet voldoen kunnen gespecifieerd worden.

Een systematische techniek voor het ontwikkelen van tools en technieken voor het debuggen van modellen is gerelateerd aan de diverse verzameling van modellen en modelleertalen die gebruikt worden bij het ontwikkelen van systemen. Het ontwikkelen van debuggers is complex, mede door de interactie tussen de semantiek van de modelleertaal (die onder andere niet-deterministisch, concurrent, tijdsgebonden, en event-verwerkend kan zijn), de verschillende noties van gesimuleerde tijd (*real-time* tegenover *as-fast-as-possible*) en de gebruiker die het uitvoeren van het model door middel van een (visuele) debugomgeving beïnvloedt. Deze thesis breidt de state-of-the-art in het ontwerpen van talen en hun geassocieerde tools uit, waarbij het mogelijk wordt gemaakt om debugoperaties toe te voegen aan eender welke modelleertaal. Hiervoor worden drie bijdragen gemaakt:

- een proces om de simulator van de taal te instrumenteren met debugoperaties;
- een architectuur, die de simulator verbindt met een (grafische) debugomgeving, waarin de domeinexpert modellen kan debuggen, gebruik makende van de abstracties van de modelleertaal;
- een workflow die toolontwikkelaars helpt om debugoperaties toe te voegen aan hun tool(s).

De technieken worden gevalideerd door ze toe te passen op een verzameling modelleertalen waarvan het gedrag sterk uiteenloopt. Deze modelleertalen zijn gekozen op basis van hun semantische eigenschappen. Dit toont aan dat de technieken generiek zijn en ook op andere modelleertalen kunnen toegepast worden. Voor elke modelleertaal wordt een verzameling van bruikbare debugoperaties gedefinieerd, die geïmplementeerd worden door het simulatiealgoritme te instrumenteren, gebruik makende van onze techniek gebaseerd op een expliciete voorstelling van de controleflow. We leggen ons toe op interactieve debugtechnieken, die gebruikers toelaten om het simulatieproces te beïnvloeden en observeren. De gebruiker benadert de debugger door middel van een (visuele) debugomgeving, die communiceert met de geïnstrumenteerde simulator. Deze debugomgeving gebruikt de abstracties van de gedebugde taal om intermediaire simulatie “snapshots” weer te geven aan de gebruiker. De debugomgeving is gegenereerd van de definitie van een *debugtaal*, die de *runtimetaal* van de

modelleertaal uitbreidt. Bovendien worden twee technieken, *live modelling* en *omniscient debugging*, geïmplementeerd voor modelleertalen. Voor deze technieken ontwikkelen we een generische workflow en technieken die het mogelijk maken ze te implementeren voor eender welke modelleertaal. We passen ze toe op representatieve voorbeelden om de techniek te valideren.

Publications

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

- HANS VANGHELUWE, DANIEL RIEGELHAUPT, SADAF MUSTAFIZ, JOACHIM DENIL, AND SIMON VAN MIERLO, *Explicit modelling of a CBD experimentation environment*, in Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS, TMS/DEVS '14, part of the Spring Simulation Multi-Conference, Society for Computer Simulation International, 2014, pp. 379–386.

  Hans came up with the idea, Daniel implemented the prototype, Simon wrote the paper, Sadaf and Joachim reviewed and modified the paper accordingly. Used in section 5.2.

- SIMON VAN MIERLO, *Explicit modelling of model debugging and experimentation*, in Proceedings of the Doctoral Symposium at MODELS'14, 2014.

  Hans provided the inspiration, Simon compiled the ideas and wrote the paper. Used in chapter 4.

- SIMON VAN MIERLO, *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.

  Simon did all the work (implementing the CBD debugger, writing the paper). Used in chapter 4 and section 5.2.

- SIMON VAN MIERLO, YENTL VAN TENDELOO, BRUNO BARROCA, SADAF MUSTAFIZ, AND HANS VANGHELUWE, *Explicit modelling of a Parallel DEVS experimentation environment*, in Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, DEVS '15, San Diego, CA, USA, 2015, Society for Computer Simulation International, pp. 107–114.

  Simon and Hans worked on the initial ideas. Yentl implemented the debugging-enhanced simulator. Simon implemented the front-end and wrote the paper. Sadaf and Bruno reviewed and modified the paper accordingly. Used in section 5.3.

- SIMON VAN MIERLO, YENTL VAN TENDELOO, BART MEYERS, JOERI EXELMANS, AND HANS VANGHELUWE, *SCCD: SCXML extended with class diagrams*, in 3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016, 2016.

Hans provided the initial idea, Joeri implemented the compiler and runtime of SCCD. Simon wrote the paper, Yentl and Bart reviewed and modified the paper accordingly. Used in section 2.3.2.

- SIMON VAN MIERLO, YENTL VAN TENDELOO, AND HANS VANGHELUWE, *Debugging Parallel DEVS*, SIMULATION, 93 (2017), pp. 285–306.

  Simon and Hans worked on the initial ideas. Yentl implemented the debugging-enhanced simulator and wrote the related work section. Simon implemented the front-end and wrote the rest of the paper. Used in section 5.3.

- SIMON VAN MIERLO, CLÁUDIO GOMES, AND HANS VANGHELUWE, *Explicit modelling and synthesis of debuggers for hybrid simulation languages*, in Proceedings of the Symposium on Theory of Modeling & Simulation, TMS/DEVS '17, San Diego, CA, USA, 2017, Society for Computer Simulation International, pp. 4:1–4:12.

  Cláudio provided the examples and the hybrid simulation algorithm. Simon and Cláudio worked on the initial ideas, and Simon implemented the debugging-enhanced hybrid simulator. Simon and Claudio jointly wrote the paper. Hans reviewed and modified the paper accordingly. Used in section 5.7.

- SIMON VAN MIERLO, YENTL VAN TENDELOO, BART MEYERS, AND HANS VANGHELUWE, *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series, Springer Int. Publishing, 2017, ch. Domain-Specific Modelling for Human–Computer Interaction, pp. 435–463.

  Bart proposed the initial idea and wrote the section on ProMoBox. Simon implemented the domain-specific language and wrote the paper together with Yentl. Hans reviewed and modified the paper accordingly. Used in section 2.2.

- YENTL VAN TENDELOO, SIMON VAN MIERLO, AND HANS VANGHELUWE, *Time- and space-conscious omniscient debugging of Parallel DEVS*, in Proceedings of the Symposium on Theory of Modeling & Simulation, TMS/DEVS '17, San Diego, CA, USA, 2017, Society for Computer Simulation International, pp. 3:1–3:12.

  Yentl implemented the solution and wrote the paper, Simon helped by writing the related work section and reviewing the paper. Hans also reviewed and modified the paper accordingly. Used in section 6.1.

- SIMON VAN MIERLO AND HANS VANGHELUWE, *Debugging non-determinism: a Petrinets modelling, analysis, and debugging tool (tool demonstration)*, in Proceedings of MODELS 2017 Satellite Events, vol. 2019, CEUR-WS, September 2017.

  Simon implemented the debugger and wrote the paper. Hans reviewed and modified the paper accordingly. Used in section 5.5.

- SIMON VAN MIERLO, ERWAN BOUSSE, HANS VANGHELUWE, MANUEL WIMMER, MARTIN GOGOLLA, MATTHIAS TICHY, AND ARNAUD BLOUIN, *Report on the 1st international workshop on debugging in model-driven engineering (MDEbug'17)*, in Proceedings of MODELS 2017 Satellite Events, vol. 2019, CEUR-WS, September 2017.

Simon, Hans, Erwan, Manuel, and Clark organized the workshop. Simon and Erwan wrote the bulk of the paper. Manuel, Clark, Martin, Matthias, and Arnaud reviewed and helped iterate over several versions of the paper. Used in chapter 1.

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

- EUGENE SYRIANI, HANS VANGHELUWE, RAPHAEL MANNADIAR, CONNER HANSEN, SIMON VAN MIERLO, AND HUSEYIN ERGIN, *AToMPM: A web-based modeling environment*, in Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), vol. 1115, CEUR, Sept. 2013, pp. 21–25.

  Raphael implemented the first prototype of AToMPM, inspired by the ideas of Hans. Simon, Conner, and Huseyin improved the prototype. Eugene wrote the paper, and Hans reviewed and modified it accordingly.

- SIMON VAN MIERLO, BRUNO BARROCA, HANS VANGHELUWE, EUGENE SYRIANI, AND THOMAS KÜHNE, *Multi-level modelling in the Modelverse*, in MULTI 2014 – Multi-Level Modelling Workshop Proceedings, 2014.

  Simon, Hans, and Bruno came up with the initial idea. Simon implemented the prototype. Simon wrote the paper. Eugene, Hans, and Thomas reviewed and modified the paper accordingly.

- BRUNO BARROCA, SADAF MUSTAFIZ, SIMON VAN MIERLO, AND HANS VANGHELUWE, *Integrating a neutral action language in a DEVS modelling environment*, in Proceedings of the 8th International Conference on Simulation Tools and Techniques, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2015, pp. 19–28.

  Bruno, Sadaf, and Hans came up with the initial idea. Bruno and Sadaf implemented the prototype, Simon helped with the front-end in AToMPM. Bruno wrote the paper. Hans, Sadaf, and Simon reviewed and modified the paper accordingly.

- JONATHAN CORLEY, EUGENE SYRIANI, HUSEYIN ERGIN, AND SIMON VAN MIERLO, *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, IGI Global, jan 2016, ch. Cloud-based Multi-View Modeling Environments, pp. 120–139.

  Jonathan implemented the multi-view architecture and wrote the paper,with help from Huseyin. Simon implemented the back-end (Modelverse) and wrote the section on it. Eugene reviewed and modified the paper accordingly.

- YENTL VAN TENDELOO, SIMON VAN MIERLO, BART MEYERS, AND HANS VANGHELUWE, *Concrete syntax: A multi-paradigm modelling approach*, in Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, New York, NY, USA, 2017, ACM, pp. 182–193.

  Simon provided the initial idea. Yentl implemented the approach in the Modelverse, and wrote the majority of the paper, while Simon and Bart helped and refined it. Hans reviewed and modified the paper accordingly.

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.

Organization of Scientific Activities

- Main organizer of the "1st International Workshop on Debugging in Model-Driven Engineering" (MDEbug'17), part of the 20th Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS'17)

Teaching

- Lab Sessions "Model Driven Engineering" course at University of Antwerp (2013-2017)
- Lab Sessions of the 6th International Summer School on Domain-Specific Modelling (DSM-TP 2015)
- Lab Sessions of the 7th International Summer School on Domain-Specific Modelling (DSM-TP 2016)
- Lab Sessions of the 8th International Summer School on Domain-Specific Modelling (DSM-TP 2017)
- Tutorial "An Introduction to Statecharts Modelling and Simulation" at SpringSim'17, 23 april 2017.

Participation in Scientific Activities

- CAMPaM Workshop 2014 (Bellairs, Barbados; 7-14 February 2014)
- 17th Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS'14) (Valencia, Spain; 28 September - 3 October 2014)

- DSM-TP Summer School 2014 (Antwerp, Belgium; 25-29 August 2014)
- CAMPaM Workshop 2015 (Bellairs, Barbados; 30 January - 6 February 2015)
- Spring Simulation Multiconference 2015 (Alexandria, VA, USA; 12-15 April 2015)
- DSM-TP Summer School 2015 (Antwerp, Belgium; 24-28 August 2015)
- 18th Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS'15) (Ottawa, Canada; 27 September - 2 October 2015)
- DSM-TP Summer School 2016 (Geneva, Switzerland; 22-26 August 2016)
- 3rd Workshop on Engineering Interactive Systems with SCXML (Brussels, Belgium; 21 June 2016)
- Spring Simulation Multiconference 2017 (Virginia Beach, VA, USA; 23-26 April 2017)
- DSM-TP Summer School 2017 (Montreal, Canada; 10-14 July 2017)
- 20th Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS'17) (Austin, TX, USA; 17-22 September 2017)

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the thesis by providing the motivation, stating the contributions, and by explaining the structure of this document.

1.1 Motivation

The complexity of (engineered) systems grows as our society increasingly depends on them. While before, mechanical systems and software systems were separated and hardly interacted, the requirements today mandate a tight integration. Software-intensive systems are characterized by a feedback loop between hardware components (which sense and act upon a physical system.) and a software controller. For example, a modern car is chiefly controlled by embedded software that continually interprets sensor values and decides an appropriate control action sent to actuators [35]. As the demand for autonomous and adaptive systems rises, a new class of system emerges which combines multiple software-intensive systems in a network: the cyber-physical system [109, 147].

With the growth in system complexity comes the need for development processes, methods, and techniques that allow system developers to successfully build such systems; concerns include safety, interoperability, and performance. *Model-Driven Engineering (MDE)* [15, 94, 180] regards models as first-class concepts in the development process. When developing systems according to the MDE process, engineers build models (abstractions) that describe the different parts of the system in (possibly graphical) modelling languages. The modelling languages provide abstractions that allow engineers to focus on *what* the system is supposed to do, instead of *how* the system is supposed to do it (which are implementation details that are important for deployment, not at design time). The models constructed during the MDE process serve a number of purposes. They allow for early Verification and Validation (V&V) using formal verification (*e.g.*, model checking), simulation, and testing techniques. They serve as documentation for the system once it is build. And, they can be used for synthesizing (parts of) the system. For hardware components, they serve as a "blueprint"; for software components, code generators can synthesize a running application.

Complexity, as well as time-to-market pressure, demand experts. These experts specialize: they are trained for one domain, and many experts work concurrently on different aspects of the system. Within MDE, *Multi-Paradigm Modelling* (MPM) [138] actively promotes this specialization: its philosophy is to model every (relevant) aspect of a system explicitly, at the most appropriate level of abstraction, using the most appropriate modelling language(s), while modelling the development process(es) explicitly. We differentiate between two different classes of modelling languages: *general-purpose modelling languages (GPMLs)* provide general abstractions that can be applied in many domains, while *domain-specific languages (DSLs)* are (often small) specialized languages that are developed to be used in one specific domain. Examples of GPMLs include the Unified Modelling Language (UML) [171], Petrinets [139], Causal Block Diagrams [33], DEVS [221], and Statecharts [73]. These languages have been researched extensively and many tools have been developed to design, simulate, analyse, test, and generate code for systems whose behaviour is specified in these languages. Building such tools is often costly, as the users demand their tools to solve increasingly difficult tasks, such as automatic verification, symbolic execution, and design space exploration. For often-used languages, this investment can be made, but for DSLs, the cost of developing tools needs to be low in order for the benefits to outweigh these costs.

Recent advances in modelling language engineering have provided language engineers with tools to quickly develop a specialized modelling language and its associated tools, including (graphical) modelling environments, simulators, and code generators. Language engineering techniques are used by language engineers that are tasked with developing a new language and its tooling. Such techniques are used to define the (abstract and concrete) syntax of languages and generate (graphical) interactive modelling environments [91]. Other techniques facilitate the definition of the modelling language's semantics by implementing a simulator for the language, or a code generator that generates appropriate implementation code. More recently, techniques focus on generating not only a design modelling language, but also a runtime language for visualizing the state of the system after and during simulation, an output language for viewing and replaying a simulation trace, and a property language for defining temporal properties the system needs to satisfy [136]. An important feature of such techniques is that the languages only slightly differ, and use domain-specific syntax to present the models and information to the language user (an engineer) to close the cognitive gap. This ensures that the engineer can focus on the essential complexity of the problem (*what* the system is supposed to do), and limiting its accidental complexity (*how* the system implements its functionality).

While modelling languages are tools that allow domain experts to more intuitively develop their models and thereby manage the development process for highly complex systems, they do not guarantee that the developed models are without *defects*. The complexity of the systems does not decrease by using a different development method: while the languages are more intuitive and specialized, the systems they can model are not simple by any means. In the past years, significant effort has been directed towards providing early verification and validation (V&V) techniques to determine whether or not a set of models fulfils a set of properties [60, 81, 156, 226]. By identifying the properties that are not satisfied, such techniques are able to discover and observe the *failures* (or *bugs*) of a system. Yet, once a failure has been observed, it is then necessary to identify why the failure occurs (*i.e.*, the defect that caused the failure), and how to modify the models to remove the cause of the failure. These two tasks constitute the core of the *debugging* activity [223]. To illustrate

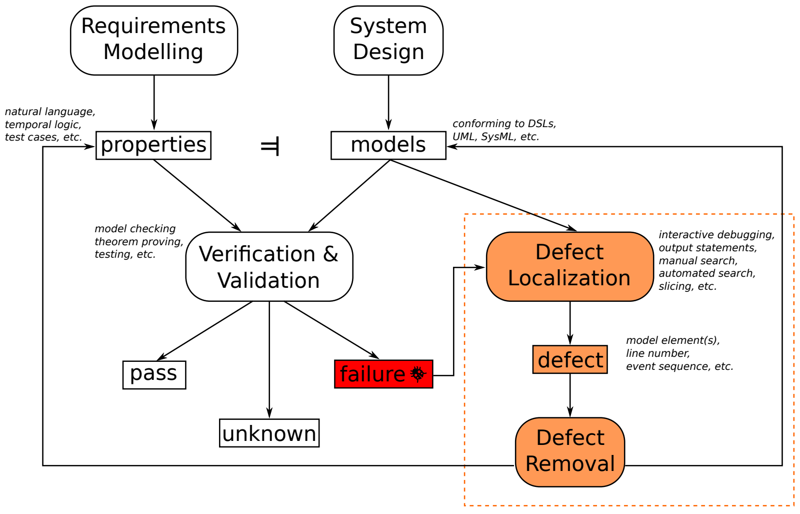

Requirements Modelling
System Design
natural language, temporal logic, test cases, etc.
properties
⊨
models
conforming to DSLs, UML, SysML, etc.
model checking theorem proving, testing, etc.
Verification & Validation
Defect Localization
interactive debugging, output statements, manual search, automated search, slicing, etc.
pass
failure
unknown
defect
model element(s), line number, event sequence, etc.
Defect Removal

Figure 1.1: The debugging process: an overview.

this activity, Figure 1.1 presents an overview of a typical system design process. First, a set of properties that the system has to satisfy is defined. Then, a system is designed as a collection of models that must satisfy these properties. To check that the properties are satisfied by the models, a wide range of verification and validation (V&V) techniques are available, such as theorem proving, symbolic execution, model checking, real-time simulation, and testing. Depending on the approach, it might be found that a property is satisfied ("pass"), not satisfied ("failure", which actually signifies a *potential* failure, since the considered V&V technique may give false positives), or that the result is inconclusive ("unknown"); this is called the *verdict*. For example, a result is inconclusive when the chosen technique cannot prove the property in a reasonable time frame. At that point, however, the *cause* of the failure (also called the *defect*) must still be identified. Either the defect is found in (one of) the design models, or the properties were wrongly specified. Once the defect(s) is (are) identified, the models or the properties have to be changed in order to remove the defect(s), after which the failure will no longer be observed.

Locating and fixing the cause of a failure can be accomplished manually given a good understanding of the models. Alternatively, a wide range of *debugging techniques* can be used to assist developers in finding the cause of the problem [223]. Since software errors have a large economic impact [145], it is imperative that such debugging tools and techniques are developed, to assist the developers as much as possible. With the growing importance of MDE techniques to develop complex systems, researchers are increasing the reliability of modelled systems by, amongst others, integrating verification and validation techniques. Yet, when it comes to models, very few debugging tools and techniques are available. To find the source of an observed failure (*i.e.*, debug) a system that is implemented using models as first-class entities, developers often have to resort to ad-hoc methods. One technique is to manually inspect the model(s) and their simulation results; since the models are defined by domain experts, they can often detect defects by relating cause (in the model) and effect (in the simulation results). A technique that is used for executable models from which code is generated consists of inspecting or debugging the code generated from models, by reusing established and well-researched program debugging techniques. This is not ideal, since the developer has to switch contexts and is required to understand the semantics of the underlying implementation language. Instead, dedicated debugging support for modelling languages and workbenches is required.

We identified the following challenges in finding a systematic way of implementing tools and techniques (*i.e.*, finding an engineering process) for model debugging, which are related to the diverse set of models and modelling languages used in a typical system development:

- The semantics of modelling languages can include non-determinism, concurrency, event-processing behaviour, dynamic structure, and much more diverse semantics natively.
- Models in different languages are often executed/simulated together, requiring their interpreters/compilers/executors to communicate. To debug such co-simulations, the debuggers for the individual languages have to be combined.
- In an MDE project, potentially many artefacts are created that might be in need of debugging, such as model transformations, language definitions, structural diagrams, etc. They need specialized debugging techniques and tools.
- Domain-specific languages have their own tools, such as workbenches, interpreters and code generators. They also require debugging support. While techniques have been developed to generate (visual) modelling interfaces from syntax definitions (*i.e.*, metamodels), it remains to be investigated whether generic techniques can be developed to generate debugging tooling as well.
- Debugging techniques for models are related to the technique used for verifying the model. Research is needed to integrate debugging techniques and tools with validation and verification techniques and tools.
- From models, often (embedded) code is generated. To debug this code (potentially in a human-, hardware-, or software-in-the-loop context), it needs to be properly instrumented, taking into account the real-time behaviour of the system.
- The source of a defect may be distributed among several models, which may each be developed in a different modelling language.

Examples of techniques that were already developed for debugging models include:

- *interactive debugging* techniques [8, 107, 128], which can be used to observe and control the execution of behavioural models in an interactive fashion (*e.g.*, using breakpoints, stepping operators, or by inspecting properties).
- (Semi-)automated fault localization, for instance using symbolic execution [175], or model slicing [16].
- Omniscient debugging techniques [19, 43], which allow their users to explore the execution trace both backwards and forwards.

There is, however, no structured approach to constructing model debugging environments based on language engineering techniques.

A note on terminology. We use the term *model simulation* as an amalgamate for *model interpretation*, *model execution*, and *model simulation*. Although we realise these are not synonyms, it serves our need: model simulation differs from model interpretation and model execution as it uses a different clock, the *simulated time*, to represent the logical time perceived by the model. Developing debuggers for such simulations is more generic, and our approach more complete, than if we only consider model interpretation or model execution. We explicitly mention if a debugger for a formalism does *not* support simulated time due to the formalism's untimed semantics, and refer to *model execution* instead.

1.2 Contributions

We contribute to the state of the art by providing a structured approach for turning modelling and simulation environments into interactive debugging environments. To achieve this, we present:

- an approach to instrument simulators with debugging support based on an explicit representation of their control flow;

- an architecture, that leverages existing components for debugging purposes, including the (graphical) modelling environment, the debugging-enhanced simulator, and any model-specific visualizations;

- a workflow that guides language engineers and tool builders to enhance their languages and tools with debugging support.

To demonstrate feasibility and validate our approach, we apply it to a set of formalisms with diverse semantics. These formalisms are carefully chosen based on an analysis of their semantic properties. For each language, we define a set of useful debugging operations and implement them by instrumenting their simulation algorithm using our technique based on an explicit model of their control flow. We focus on interactive debugging techniques which allow users to control and observe the simulation process. The user controls the debugger through a (graphical) interface which communicates with the debugging-enhanced simulator. These interfaces use the abstractions provided by the debugged language to present intermediate states of the simulation to the user. The interfaces are generated from a definition of a *debugging language*, which extends the *runtime language* of the modelling language.

Finally, we explain how two advanced debugging techniques can be (efficiently) implemented: omniscient debugging (which allows user to step back in a simulation trace) and live modelling (which allows users to modify the design model during simulation). Again, we provide a generic workflow and techniques for implementing these operations for any modelling language, and apply these techniques to representative examples to demonstrate feasibility.

1.3 Structure

Chapter 2 provides background information on the techniques and formalisms on which the techniques developed in this thesis depend. Chapter 3 reviews the state-of-the-art in debugging techniques, both for debugging code, as well as for debugging models. Chapter 4 extends the state-of-the-art in language engineering and modelling tool construction with techniques, an architecture, and a workflow for systematically augmenting modelling environments and simulators with debugging support. Chapter 5 evaluates these new techniques by applying them to a set of formalisms with diverse semantics to demonstrate feasibility. Chapter 6 discusses the implementation of two advanced model debugging techniques, transposed from program debugging: omniscient debugging and live modelling. Chapter 7 concludes the thesis and points towards possible directions for future work.

Chapter 2

Background

This chapter provides background to the thesis. It presents a number of concepts, methods, and techniques that we will use to develop a generic technique to construct debugging environments for modelling languages. First, Section 2.1 explains *Multi-Paradigm Modelling* (MPM), a philosophy towards the modelling of complex, software-intensive systems. MPM is central to this thesis, as the implemented debugging tools and techniques are an important addition to the tool set used for modelling systems with MPM techniques, and because we use MPM techniques for developing those tools. Second, Section 2.2 explains how modelling languages and their tooling are developed in a structured, repeatable, and efficient way. These language engineering techniques are important towards the reliable implementation of advanced modelling tools and environments. Third, Section 2.3 focuses on one particular class of systems: timed, reactive, autonomous systems, whose behaviour is difficult to express using standard, imperative programming languages. Instead, the section explains how the Statecharts modelling language is used to more effectively develop such systems, as we will use Statecharts to model the behaviour of debugging-enhanced simulators. Last, Section 2.4 presents an assumed modelling and simulation workflow that engineers can use for developing systems, listing the artefacts that are created. This motivates the need for debugging techniques and tools to assist engineers when they revise their systems as failures are detected. It also introduces the artefacts that are created as part of the modelling process, which potentially can be (re)used for debugging purposes.

2.1 Multi-Paradigm Modelling

The complexity of (engineered) systems grows as our society increasingly depends on them. While before, mechanical systems and software systems were separated and hardly interacted, the requirements today mandate a tight integration. Software-intensive systems are characterized by a feedback loop between hardware components (which sense and act upon a physical system.) and a software controller. For example, a modern car is chiefly controlled by embedded software that continually interprets sensor values [35]. As the demand for autonomous and adaptive systems rises, a new class of system emerges which combined multiple software-intensive systems in a network: the cyber-physical system [109, 147].

With this growth in system complexity comes the need for development processes, methods, and techniques that allow system developers to successfully build such systems; concerns include safety, interoperability, and performance. *Model-Driven Engineering (MDE)* [15, 94, 180] regards models as first-class concepts in the development process. When developing systems according to the MDE process, engineers build models (abstractions) which describe the different parts of the system in a (graphical) modelling language. These models serve a number of purposes. They allow for early Verification and Validation (V&V) using simulation and testing techniques. *Modelling and Simulation (M&S)* [208] techniques focus on virtual experimentation to "get it right the first time": even before any physical component of the system is built, accurate simulations can predict its behaviour. For hardware components, models serve as a "blueprint"; for software components, code generators can synthesize a running application. Such techniques are possible only if modelling languages have precisely defined syntax and semantics.Models document the system by making explicit assumptions, requirements, and design choices of the system.

Developing complex, software-intensive systems demands experts. These experts specialize: they are trained for one domain. Within MDE, *Multi-Paradigm Modelling (MPM)* [138] actively promotes this specialization: its philosophy is to model every (relevant) aspect of a system explicitly, using the most appropriate modelling language(s), while modelling the development process(es) explicitly. Tailored modelling languages allow an engineer to focus on *what* the solution to a problem is, not *how* it is implemented. The solution is developed in a notation that is most familiar to the engineer.

2.2 Building Modelling and Simulation Tools

Many *General-Purpose Modelling Languages (GPMLs)* have been developed, a few well-known examples include Statecharts [73], DEVS [221], and CBD [33]. Such languages can model a large set of systems due to their generality. Moreover, as a language becomes popular, its user base grows and mature tooling is developed for more effectively developing, simulating, and verifying models in these languages. In certain cases, however, more specialized languages are needed. *Domain-Specific Modelling (DSM)* [91] in particular makes it possible to specify models in a *Domain-Specific Modelling Language (DSML)*, using concepts and notations of a specific domain. DSMLs are an integral part of the MPM philosophy, where they seamlessly integrate with GPMLs and other DSLs. The goal is to enable domain experts to develop, understand, and verify models more easily, without having to use concepts outside of their own domain. DSMLs allow the use of a custom visual (graphical or textual) syntax, which is closer to the problem domain, and therefore more intuitive. There is, however, a cost involved: a language engineer needs to create the DSML, which includes defining its syntax, providing the mapping between the problem domain and the solution domain, and building the necessary tooling: (graphical) modelling environments, verification and validation tools, simulators, and code generators. Recent advances in modelling language engineering have provided language engineers with tools to quickly develop a specialized modelling language and its associated tools.

A modelling language, or formalism, is fully defined [97] by:

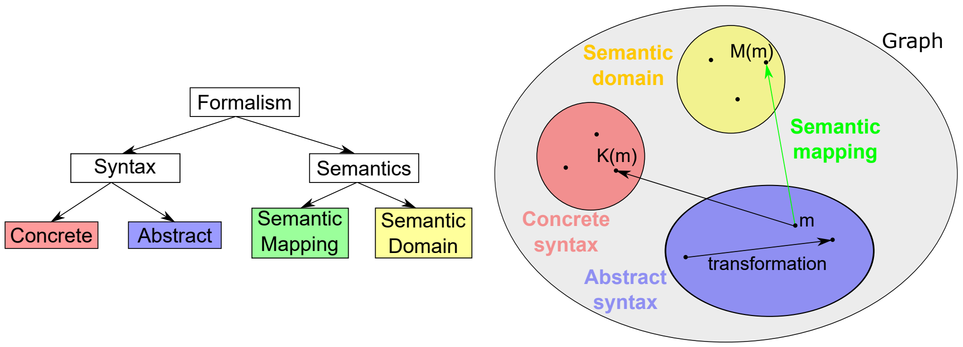


Figure 2.1: Defining a formalism.

1. Its **abstract syntax**, defining the language constructs and their allowed combinations. This information is typically captured in a metamodel for graphical languages, or a grammar for textual languages.
2. Its **concrete syntax**, specifying the (often visual) representation of the different constructs. This visual representation is typically graphical (using icons), or textual. Others, such as sound, are possible as well [203].
3. Its **semantics**, defining the meaning of models created in the domain [77]. This encompasses both the **semantic domain** (*what* is the meaning of the language), and the **semantic mapping** (*how* to give the models in the language meaning).

For example, $1 + 2$ and $(+\ 1\ 2)$ can both be seen as textual concrete syntax (*i.e.*, visualizations) for the abstract syntax concept "addition of 1 and 2" (*i.e.*, the abstract concept represented by the textual syntax). The semantic domain of this operation is the set of natural numbers (*i.e.*, what the expressions evaluates to), with the semantic mapping being the execution of the operation (*i.e.*, how the expression is evaluated). Therefore, the semantics, or "meaning", of "addition of 1 and 2" (represented by $1 + 2$ or $(+\ 1\ 2)$) is 3.

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. In the next subsections, we survey language engineering methods for each aspect of a formalism's definition. We first explain how the syntax of a language is defined, and how a (graphical) modelling environment is synthesized from this definition (Section 2.2.1). Then, we explain model transformations, which manipulate models by rewriting their structure, possibly translating models to a different language (Section 2.2.2). Model transformations are used in the next two subsections: Section 2.2.3 explains how semantics are defined for a modelling languages, and Section 2.2.4 explains how workflows for developing systems are modelled, including an explicit representation of the produced artefact's languages and their relations.

2.2.1 Syntax of Modelling Languages

Syntax defines whether elements and constructs are valid in a specified language or not. It does not, however, concern itself with what the constructs mean. With syntax only, it would

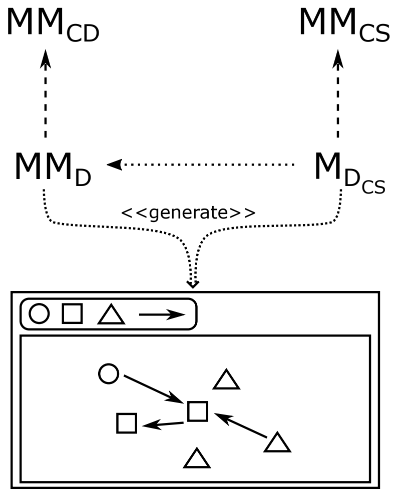


Figure 2.2: Generating a modelling environment for a design language *D*.

be possible to specify whether a construct is valid, but it might have undefined or invalid semantics. A simple, textual example is the expression $\frac{1}{0}$. It is perfectly valid to write this, as it follows all structural rules: a fraction symbol separates two recursively parsed expressions. Its semantics is undefined, however, since it is a division by zero.

The *abstract syntax* definition of a language specifies its constructs and their allowed combinations. Such definitions are captured in a grammar, in rules, or in a metamodel, which itself conforms to a metametamodel [105]. Most commonly, the metametamodel is similar to UML Class Diagrams. The metametamodel used in this thesis whenever we present a metamodel allows a language engineer to define classes, associations between classes (with incoming and outgoing multiplicities), and attributes.

While the abstract syntax reasons about the allowable constructs, it does not state anything about how they are presented to the user. In this way, it is distinct from textual grammars, as they already offer the keywords to use [97]. It merely states the concepts that are usable in the domain. The *concrete syntax* definition of a model specifies how elements from the abstract syntax are (visually) represented. The relation between abstract and concrete syntax elements is also modelled and made operational with *parsers* (deducing the abstract syntax from a concrete syntax model) and *pretty printers* (deducing the concrete syntax from an abstract syntax model). Syntax-directed editing environments only allow users to create syntactically valid models using concrete syntax, in which case parsing and pretty printing is not necessary, since the editing of models encompasses both creating the concrete syntax and associated abstract syntax element(s). A language has one abstract syntax definition, but can have multiple concrete syntax definitions, and a domain-specific environment can allow a user to switch between representations. The definition of the concrete syntax is a determining factor in the usability of the DSML [9].

The language definition, consisting of the abstract syntax definition and concrete syntax definition, can be used by a meta-modelling environment (such as AToMPM [192]) to generate a syntax-directed modelling environment for that language. This is visualized in Figure 2.2, where a metamodel for a language *D* (MM_D) (conforming to the Class Diagrams metamodel (MM_{CD})) and a definition of its concrete syntax ($M_{D_{CS}}$, conforming to the Concrete Syntax formalism (MM_{CS})) are used to generate a graphical modelling environment. The environment has a toolbar with icons for each language element (in this case, circles, rectangles, triangles, and arrow). By using this toolbar, the modeller can instantiate elements of the language on a canvas, creating a valid design model.

A metamodel and concrete syntax definition do not define the semantics of a modelling language. Nor do they specify how the runtime state of a system is to be visualized. For that, recent techniques allow language engineers to turn their design language into several languages used for execution, simulation, and analysis [136]. Before we discuss those, we explain model transformation, a technique that can be used to define the semantics of a modelling language.

2.2.2 Model Transformation

Model transformation has been called the heart and soul of MDE [181]. When using multiple languages to model a system in, model transformations can define relations between languages and models expressed in those languages. This is useful, since models might be related: for example, one model might be a refinement of another model, or one model might encode the semantics of another model. Model transformations can be categorized [134]. *Endogenous* transformations transform models in a language to models in the same language, while *exogenous* transformations transform models in a language to models in another language; *in-place* transformations modify the model in place, while *out-place* transformations create a new model.

Model transformations can be specified in numerous ways; we mention three different paradigms. A transformation can be specified manually using an imperative programming language. The code of the transformation needs access to the data structures of the model (the graph structure) to traverse it, query it, and make modifications to it. Such methods regard models as graph-like structures in-memory and do not take advantage of the extra information added by the conformance relation between a model and its metamodel. They are easy to implement for programmers with a background in imperative programming languages, however. On the other side of the spectrum are *Triple Graph Grammars (TGGs)* [179], a declarative approach to model transformation. TGGs allow a modeller to specify relations between models using *patterns*. These relations are automatically made operational and maintained by a transformation engine. For example, such an engine might translate source models to target models automatically, or maintain consistency between two models in both directions. The modeller has no influence on the engine and cannot control, for example, the order in which rules are executed. A hybrid approach allows users to model transformation rules (based on patterns) declaratively and to model the scheduling in an appropriate language. MoTiF [191] is such a language: it provides primitives to schedule rules in an explicit control flow. Rules are modelled explicitly as well. A rule consists of:

- exactly one positive precondition pattern (a *Left-Hand-Side (LHS)*) that, when matched, actives the rule;
- a number of negative precondition patterns (*Negative Application Conditions (NACs)*) that, when matched, deactivate the rule;

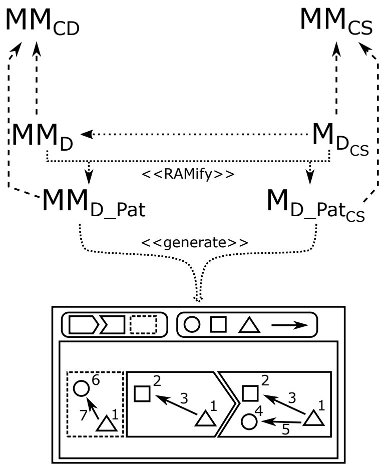


Figure 2.3: Generating a modelling environment for a RAMified design language *D_PAT*.

- a postcondition pattern (a *Right-Hand-Side (RHS)*), that specifies how the matched elements are to be rewritten.

The patterns are models themselves; and, following the MPM philosophy, their structure should be similar to the models to be transformed. After all, patterns specify that a part of a model (a number of elements, relations, and attribute values) need to be matched, or specify how elements in a model are to be altered (by removing or adding elements and relations, or by changing their attribute values). An automated approach that constructs such languages from the original definition of the language was introduced by Kühne et al. [106]. They propose to *Relax*, *Augment*, and *Modify* (*RAMify*) the design metamodel of the language to arrive at a pattern language for use in the *LHS*, *RHS*, and *NACs* of a rule.

Figure 2.3 shows an example of RAMification on our example language *D*: an automatic transformation transforms its metamodel and concrete syntax definition to a pattern metamodel and accompanying concrete syntax definition. From that language definition, a new modelling environment is generated for describing patterns. In this environment, new concepts become available: we can model rules, in the form of a left-hand-side pattern (represented by an "arrow" pointing right), a right-hand side pattern (denoted by an "inverted arrow"), and a set of negative application conditions (represented by dotted rectangles). An example rule is shown: it transforms a triangle connected to a square, by adding a circle, but only if no circle is connected to the triangle. This rule needs to be scheduled. For example, we might want to find all matching pairs of squares and triangles and execute the rule for all of them. Or, we might match a pair of squares and triangles and rewrite them, until no more matches are found. The subtle difference between the two is that in the second case, a rewrite might cause a match that was found in the first case to not be found (since a circle was connected to a triangle). MoTif is a possible scheduling language, but others are possible. Action language, for example an imperative programming language, is well-suited: its *while-* and *for*-loops naturally allow for a schedule to be encoded.

2.2.3 Semantics of Modelling Languages

Since the syntax only defines what a valid model looks like, we need to give a meaning to the models. Even though models might be syntactically valid, their meaning might be useless or even invalid (as pointed out above, the division $\frac{1}{0}$ is syntactically valid, but semantically invalid).

It is possible for humans to come up with intuitive semantics for the visual notations used (*e.g.*, an arrow between two states in a finite-state automata model means that the state changes from the source to the destination if a certain condition is satisfied). There is, however, a need to make the semantics explicit for two main reasons:

1. Computers cannot use intuition, and therefore there needs to be some operation defined to convey the meaning to the machine.

2. Intuition might only take us that far, and can cause some subtle differences in border cases. For example, the specification of Java threads was ambiguous and difficult to interpret, leading to inconsistent implementations [166]. Having semantics explicitly defined makes different interpretations impossible, as there will always be a "reference implementation".

Semantics consists of two parts: the domain it maps to, and the mapping itself. 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 of its own can act as a semantic domain. The choice of semantic domain depends on which properties need to be conserved. For example, DEVS [221] can be used for simulation, Petrinets [139] for verification, Statecharts [73] for code synthesis, and Causal Block Diagrams (CBD) [33] for continuous systems using differential equations. A single model might even have different semantic domains, each targeted at a specific goal. The semantic domain might also be the language itself, in which case the semantic mapping implements the semantics of the language without relying on the semantics of a target language.

We focus on the two categories of semantic definitions [225]:

1. **Translational semantics**, where the semantic mapping 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 (again, either translational or operational), meaning that the semantics is "transferred" to the original model.

2. **Operational semantics**, where the semantic mapping 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 defined completely in the problem domain (the semantic domain is equal to the original language), making it suitable for use by domain experts. A disadvantage of this approach is that the semantics are difficult to optimize, since this requires knowledge of the implementation and underlying platform, which domain experts do not necessarily have. The simulator can be implemented using model transformations, or in action language.

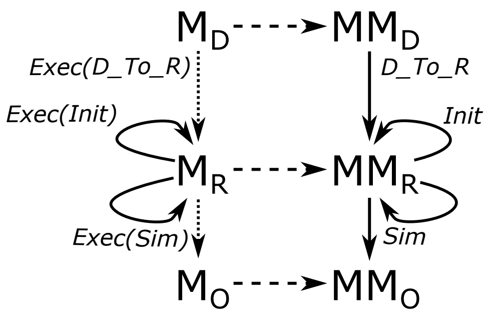


Figure 2.4: Operational semantics.

Figure 2.4 illustrates how operational semantics work. Three languages are involved: a *design language*, a *runtime language*, and an *output language*; their metamodels MM_D, MM_R, and MM_O, as well as the operations between them are shown in the figure:

- The design language is used to model the static structure of the system; this does not mean that the behaviour of the system is necessarily static in nature, but it means that the design model does not contain any runtime information.

- The runtime language is an extension of the design language and adds runtime attributes to the design language. Conforming models are "snapshots" in the runtime trace of a system. A runtime model is obtained from a design model by executing the *D_To_R* transformation. The initial runtime state of a model is obtained by executing the *Init* transformation. This operation initializes the runtime attributes of the model.

- The output language models output traces, verification results, or simulation results. Its structure depends on which questions the models in the language need to answer. An output model is obtained by executing the *Sim* operation, which represents simulation, interpretation, or execution of the runtime model. During simulation, the runtime state of the model is updated in consecutive "snapshots" (models conforming to the runtime language). If the output language contains a full state trace, these snapshots are stored and communicated to the user at the end of simulation.

Translational semantics are another way of defining semantics for a modelling language. Traditionally, denotational semantics allow to formalize the semantics of programming languages in terms of mathematical objects (the *denotations*). Such semantics are referred to as Scott-Strachey semantics for programming languages [189]. We interpret denotations more broadly, and allow any model element in a formalism (not only the mathematical formalism) to serve as a denotation. In that context, the more general translational semantics are defined by building a transformation which transforms any conforming design model into a conforming design model of a language with known semantics. This technique is often useful in case a similar language exists with mature tooling. Instead of reinventing the wheel, the language designer exploits the similarities between his language and the target language, and reuses the already existing infrastructure.

Figure 2.5 illustrates how translational semantics work. We assume a target language with known semantics, whose design language, runtime language, and output language are described by the metamodels MM_{Trg_D}, MM_{Trg_R}, MM_{Trg_O}. We also assume semantics

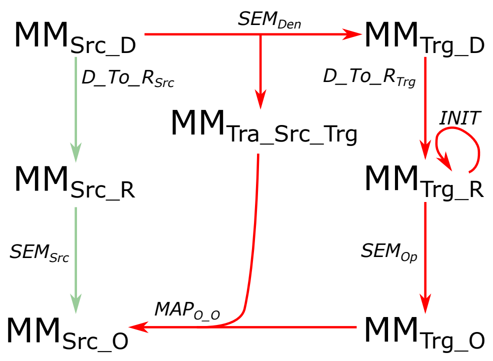


Figure 2.5: Translational semantics.

are defined for this language operationally as discussed above. The source language is defined by a design language MM_{Src_D}, a runtime language MM_{Src_R} and an output language MM_{Src_O}. A number of operations are defined between these languages:

- A design model conforming to the source metamodel MM_{Src_D} is transformed to a design model in the target language MM_{Trg_D}. This operation is a total function: any valid model in the source language can be transformed to a model in the target language. As a side effect of this transformation, a traceability model (conforming to $MM_{Tra\_Src\_Trg}$) is created which contains information on how elements in the source model are related to models in the target model.

- The target language has its semantics defined operationally: as discussed above, an output model conforming to MM_{Trg_O} is created as a result of simulating the model. For this discussion, we assume the operational semantics is a black box: no intermittent updates to the runtime model are visible.

- To obtain a model in the source output language MM_{Src_O}, the target output model is translated through the Map_{O_O} transformation.

Translational semantics allow to reuse existing infrastructure to simulate models. If done correctly, however, a user of the source language needs not know that the semantics are defined translationally: from the outside, a request for simulating a model in the source language results in an output model just the same as it would do if the semantics were defined operationally. The view of the modeller is represented by the light green arrows between the source languages:

$$SEM_{Src}(M_{Src}) = (MAP_{O\_O} \circ SEM_{Op} \circ INIT \circ D\_To\_R_{Trg} \circ SEM_{DEN})(M_{Src})$$

We do not consider the case where the user wants to view intermittent states of the simulation at the source level for now. This will be addressed in a later chapter, when we define debugging for a domain-specific language whose semantics are defined translationally.

2.2.4 Modelling Workflows with FTG+PM

Process modelling is a widely used technique on the business level of a project. Business process modelling formalisms, however, fall short of capturing the essence of the engineering nature of complex system development. To model workflows and the relation between formalisms in the context of MPM, a Formalisms Transformation Graph and Process Model (FTG+PM) language is designed specifically for depicting model-driven development processes and offers appropriate abstractions. The first version of a formalism transformation graph was presented by Vangheluwe and Vansteenkiste [209]. Such models represent relations between formalisms explicitly; relations represent a (semantic) correspondence between the languages and are made operational using model transformation. Later, the graph was extended with an explicit workflow model [119, 120], and a case study was performed in the automotive domain [140]. The language allows the modelling of an engineering workflow whose deliverable is a set of (software) artefacts, and allows the modeller to relate each artefact to the language it is specified in, as well as to specify transformations between the used languages. The process model part of an FTG+PM model represents a workflow consisting of activities that require user input (manual activities) and activities that do not require user intervention (automatic transformations). The formalism transformation part describes a formalism transformation graph consisting of a number of formalisms and (automatic/manual) transformations between them. The workflow and formalism transformation graph are related: activities in the workflow are explicitly typed by transformations in the formalism transformation graph, while the artefacts created in the workflow are typed by the formalisms in the formalism transformation graph.

Models in the FTG+PM language can be used for various purposes. By explicitly representing workflows (for engineering processes) they can be reused for developing several systems, used to extract traceability information, and used for certification of regulatory requirements. Workflows can furthermore be analysed and optimized by reasoning over the complete transformation chain. And last, by not only modelling the workflow, but also the languages of the produced artefacts, the workflow can be enacted in a visual modelling environment. The model then guides the modeller by opening language-specific model editors for manual tasks and executing automatic transformations when it is required. FTG+PM languages are an essential part of the MPM approach, making it a repeatable and analysable process for developing complex engineered systems.

Figure 2.6 shows an example FTG+PM model. It models a workflow for developing systems using the Statecharts language (discussed in the next section). On the left (formalism transformation graph) side, five languages are involved: a language for modelling textual requirements (Text. Req.), the Statecharts language for modelling the system, a trace language (TraceLang), a language for representing boolean values (Boolean), and the Java programming language. Between these languages, several (automatic or manual) transformations are modelled for gathering requirements, modelling and revising the system, testing and simulating the system, and generating executable code. On the right (process model) side, the workflow orchestrates the execution of these transformations as activities: it *orders* them. First, the requirements need to be defined, resulting in an artefact that conforms to the Text. Req. language. Then, the system is modelled, resulting in a first version of the design model conforming to Statecharts. This model is subsequently simulated and tested in orthogonal branches of the workflow. In the simulation branch, an input trace is used by the simulator, and the output of the simulation is checked to be

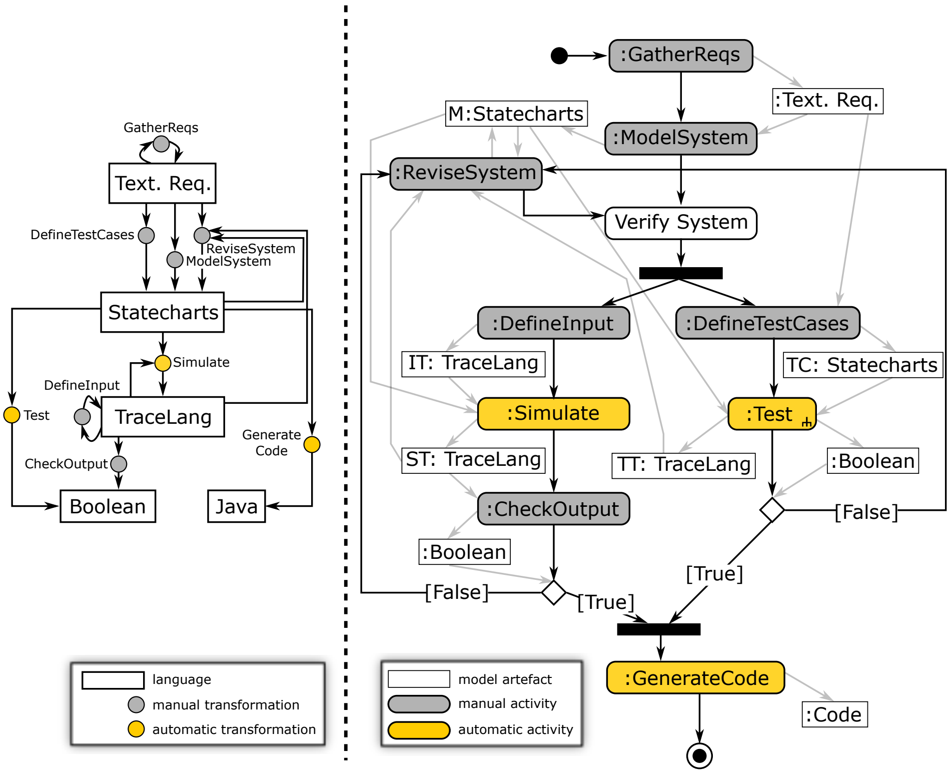


Figure 2.6: An example FTG+PM model for developing systems using the Statecharts language.

correct. In the testing branch, test cases are defined and run on the model, again checking whether the system behaves as expected. Based on the simulation and test results, the system is iteratively refined. Last, when simulation and testing return satisfactory results, code is generated that executes the system. We use FTG+PM models whenever we describe workflows in this thesis.

2.3 Modelling System Behaviour with Statecharts

Timed, reactive, autonomous systems are challenging to design and develop. Behaviour is inherently concurrent—the autonomous behaviour of the system has to be interleaved with its handling of external (coming from the environment) events. The environment is non-deterministic, since we cannot predict when external events will arrive. And, while operating systems offer threading interfaces natively, programming with threads, locks, and semaphores quickly becomes unmanageable [108]. Instead, such behaviour is better expressed at an abstraction level which hides these implementation details—one which focuses on the essential complexity. Statecharts was introduced by Harel [73] as a graphical, topological formalism for describing complex system behaviour and is an appropriate language for modelling timed, reactive, autonomous behaviour. In Section 2.3.1

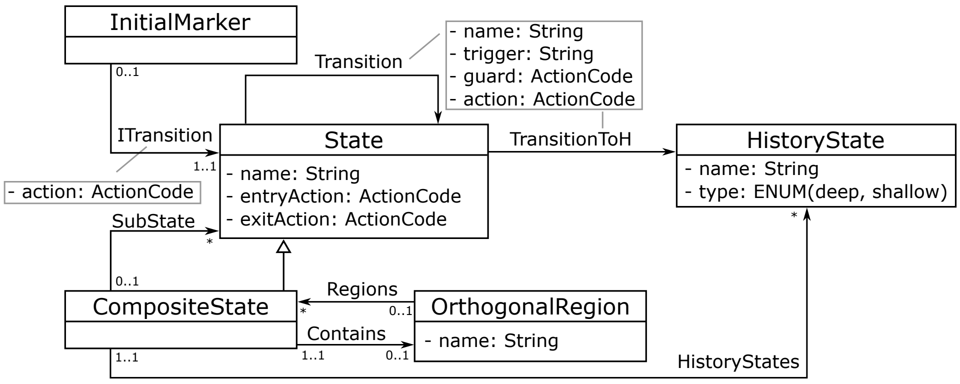


Figure 2.7: The metamodel for the Statecharts language.

we explain the syntax and semantics of Harel's Statecharts formalism. In Section 2.3.2, we expand upon his work and introduce SCCD, a language which adds dynamic structure to the semantics of Statecharts.

2.3.1 Harel Statecharts

Statecharts is an extension of state machines and state diagrams with hierarchy, orthogonality, and broadcast communication. Its metamodel is shown in Figure 2.7. A Statecharts model consists of states, that explicitly encode a set of discrete states the system can be in and transitions between those states that model the dynamics of the system. In the next subsections, we explain each language feature.

States

A state is the basic building block of a Statecharts model. It has the following attributes:

- A unique *name* within its enclosing scope.
- An optional *entry action* that is executed when the state is entered.
- An option *exit action* that is executed when the state is exited.

States can be composed hierarchically in composite states (which, when they are active, have exactly one active child state), as well as orthogonally in orthogonal regions (which are all active when their parent state is active). Within each composite state, exactly one state is the default state. When the composite state is entered, its default state is entered as well (except when a history state is present in the composite state, see below). An orthogonal region is composed of multiple composite states: upon entering the orthogonal region, all its child states are entered as well. History states are used to remember the active child state of a composite state. When the composite state is re-entered, the state that was originally active is restored. History states can be shallow, remembering only the direct child of its parent state that was active, or they can be deep, in which case they remember all active descendant states.

Transitions

Transitions between states model the dynamics of the system. When a transition is executed, its source state (and its descendants) is (are) exited, and its target state (and its descendants) is (are) entered. An algorithm determines the least common ancestor of the source and target states, and also exits any states up to (but excluding) the least common ancestor, and enters any states down from the least common ancestor to the target state. A transition is *triggered* by an external event (coming from the environment), an internal event that was raised by an orthogonal component, or a timeout. This is how time is incremented in Statecharts: the clock is synchronized either with the timeout event, or with the time at which an external interrupt arrives. An optional *guard* specifies an additional runtime condition that must be satisfied in order for the transition to be enabled. An optional *action* is executed when the transition is executed.

Actions

As part of the transition's execution, or when entering or exiting a state, an action can be executed. An action is specified in an action language with imperative constructs for modifying runtime variables. Actions can also *raise* an event. The *scope* of the raised event is either local, in which case it can trigger transitions in components orthogonal to the component in which the event was raised, or the event can be raised on an output port, in which it can be "sensed" by the environment. Actions are assumed to take zero time, although the real delay of executing the action can be non-zero (especially in the case of heavy computations). The time as perceived by the Statecharts model, however, does not increase. This can cause the Statecharts clock to lag behind the wall-clock time, in which case the runtime has to resynchronize the clocks when the computational load is lower.

Additional Structures

We assume that a Statecharts model is accompanied by:

- a memory component that stores variables and their values;
- a non-modal component, consisting of functions that read and write the values in the store;
- an interface definition to interact with its environment in the form of a set of input and output *ports*.

We describe a structure $S =< SC_S, Mem_S, NonM_S, X_S, Y_S >$ where:

- SC_S is the Statecharts model describing the behaviour of the system using the above concepts. A formal description of Statecharts can be found in [76].

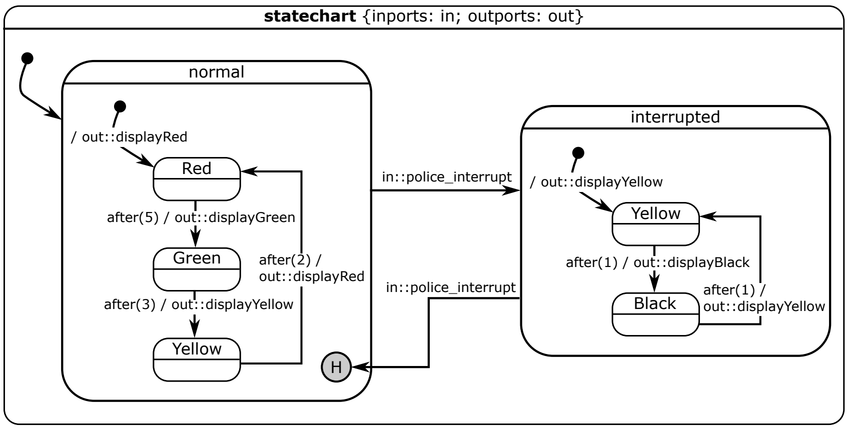

statechart {inports: in; outports: out}
normal
/ out::displayRed
Red
after(5) / out::displayGreen
Green
after(2) / out::displayRed
after(3) / out::displayYellow
Yellow
H
in::police_interrupt
in::police_interrupt
interrupted
/ out::displayYellow
Yellow
after(1) / out::displayBlack
Black
after(1) / out::displayYellow

Figure 2.8: An example Statecharts model.

- Mem_S is a store that maps variable names onto their value. The values that can be stored depend on the data types available; we will not go into detail, and assume the Statecharts model is compiled to a general-purpose language such as Python, and can use its data types.
- $NonM_S$ is a set of functions ('non-modal' functionality) that modify the data store Mem_S. These functions are called in the actions modelled in the Statecharts.
- $X = \{(p, v) | p \in \mathit{IPorts}, v \in X_p\}$ describes the input of the model. *IPorts* is a set of input port names. Each port p has an associated set of possible input events X_p.
- $Y = \{(p, v) | p \in \mathit{OPorts}, v \in Y_p\}$ describes the output of the model. *OPorts* is a set of output port names. Each port p has an associated set of possible output events Y_p.

Statecharts is a deterministic formalism, and different language variants exist, corresponding to different deterministic orderings and interleavings of events and transitions. We assume the STATEMATE [75] semantics, but other reasonable choices are possible [51, 74].

Example

Figure 2.8 shows an example Statecharts model: the behaviour of a traffic light. It has one input port *in* and one output port *out*. In the Statecharts model, events received on input ports or sent on output ports by the Statecharts model are prepended by *<portname>::* where *<portname>* is the name of the port. The traffic light is initialized in the *normal* state (denoted by a black dot with an outgoing arrow). Within the *normal* state, the traffic light cycles through its *Red*, *Green*, and *Yellow* states. When a transition is taken, an event is raised on the output port to communicate what the current colour of the traffic light is. The traffic light can be interrupted by a *police interrupt*. Whatever substate of the *normal* state it is in, it transitions to the *interrupted* state. Within that state, it cycles through its *yellow* and *black* state, raising appropriate events on its output port. The police can resume the traffic light, at which point the state it was in before the interrupt is restored (using a history state).

We use Statecharts extensively throughout this thesis to model (parts of) the behaviour of debugging environments.

2.3.2 SCCD: Extending Statecharts with Dynamic Structure

While Statecharts is an appropriate formalism for describing the timed, reactive, autonomous behaviour of systems, it does not allow to model a system with dynamically changing structure. In many software applications, objects are continuously created and destroyed. If we want to describe the software system's behaviour using Statecharts, we would not be able to: we would have to resort to an object-oriented programming language that connects compiled Statecharts models at runtime and manages multiple instances of them. This is not ideal, as that program code is hand-crafted, which means it is prone to errors.

The SCCD formalism extends Statecharts with the concepts of the Class Diagrams formalism (classes and relations), which model structure, and associates with each class a definition of its behaviour (in the form of a Statecharts model).

Classes

Classes are the main addition of the SCCD language. They model both structure and behaviour. Structure is modelled in the form of attributes and relations with other classes, effectively encapsulating the runtime data of the application. Behaviour is modelled in the form of methods, which access and change the values of attributes of the class by executing statements modelled in an action language, and a Statecharts model, which constitutes the "modal" part (*i.e.*, the control flow) of the class. Compared to Statecharts, the additional structures Mem_S and $NonM_S$ are now encapsulated in class attributes and methods, respectively. At runtime, an object can be created and deleted, followed by the invocation of, respectively, the constructor and the destructor. The relationships modelled between classes are instantiated at runtime in the form of links. They serve as communication channels, over which objects can send and receive events. There is exactly one default class, of which an instance is created when the system is started by the runtime.

Relationships

Classes can have relationships with other classes. An *association relation* is defined between a source class and a target class, and has a name. An association has a multiplicity, defined as a minimal cardinality $c_{min} \in \mathbb{N}$ and a maximal cardinality $c_{max} \in \mathbb{N}_{>0} \cup \{\infty\}$. By default, $c_{min} = 0$ and $c_{max} = \infty$. They control, at runtime, how many instances of the target class have to be minimally associated to each instance of the source class, and how many instances of the target class can be maximally associated to each instance of the source class, respectively. Each time an association is instantiated, it results in a uniquely identified link between the source and target object which can be used, for example, to send events. An *inheritance relation* results in the source of the relation to inherit all attributes and methods from the target of the relation. Specialisation of the superclass's behaviour, however, is not supported.

Event Scopes

With the addition of a public input/output interface using ports, as well as classes and associations, comes the need for event scoping. Compared to Statecharts scoping of events, SCCD adds the ability to transmit events to class instances. In particular, the following scopes are defined:

- `local`: The event is only visible for the sending instance (the default behaviour of Statecharts).
- `broad`: The event is sent to all currently running instances.
- `output`: The event is sent to an output port.
- `narrow`: The event is narrow-cast to specific instances only that are connected to the sending instance with a link.
- `cd`: The event is processed by the object manager. See the next section for more details.

Runtime Behaviour

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. We list them below, including the parameters that have to be sent as part of the event:

- **create_instance**(*association_name*, *class_name*, . . .): creates a new instance, if it is allowed (*i.e.*, no constraints would be violated). The newly created instance is always associated to its creator (the instance that sent the event). The first parameter is the name of the association that should be instantiated to create a link between the parent and its newly created child. The second parameter is the name of the class that needs to be instantiated. This should be the class that is defined as the target of the association, or one of its subclasses. Any subsequent parameters are interpreted as arguments to the constructor of the new instance. If creation succeeds, a reply event is sent to the requester containing the unique identifier of the link created between the creator and the new object. If creation failed, an error event is sent instead.

- **delete_instance**(*link_ref*): deletes the instances specified by the link reference. The link reference is evaluated in the context of the instance that sent the event and should result in a set of link identifiers. The target objects of these links are deleted, as well as any links for which these objects are the source or target, as long as no multiplicity constraints are violated. The object manager sends an event to the requester when deletion was successful. If deletion failed, an error event is sent instead.

- **start_instance**(*link_ref*): starts the execution of the Statecharts model of the instances specified by the link reference.

- **associate_instance**(*source_ref*, *association_name*, *target_ref*): creates an instance of an association (a link) between two existing instances. The source and target references are evaluated to two sets of instances, and each instance in the first set is connected using the specified association with the instances in the second set.

The object manager is also responsible for keeping track of which objects will execute a transition. It ensures a proper scheduling between objects based on events they receive from the environment or from each other. The Statecharts models of the objects execute according to Harel Statecharts; in essence, each object has its own Statecharts executor, and all executors are globally synchronized to ensure input and messages between objects are properly handled.

2.4 Modelling and Simulation Workflow

When modelling and simulating complex systems, a number of artefacts are typically created. While we do not discuss different M&S workflows in-depth (this can be found in other works, such as [32]), we present a generic workflow and assume that a number of artefacts are created by the domain expert.

We present a (high-level) overview of the presumed modelling and simulation workflow in Figure 2.9 as an FTG+PM model. The domain expert is expected to produce three artefacts: a model of the system (modelled in a design language), a set of experiments (which could be modelled in an experiment language such as SESSL [52]), and an (optional) domain-specific visualization. This last artefact can, for example, be used for systems that exhibit agent-like behaviour, where each agent is assigned a visual representation, and the domain expert can follow and potentially influence the simulation by interacting with the visualization. We assume in this thesis that the domain expert models the behaviour of the visualization in Statecharts. Although this assumption does not always hold, Statecharts is an appropriate formalism to specify the "modes" of the visualization in an explicit control flow model [96]. Therefore, it is reasonable to require the modal behaviour to be specified using this formalism. From this model, code is generated and combined with an existing visualization library to build an executable model-specific visualization interface. The visualization and simulation model are strictly separate; the visualization can only interact with the model through its interface (and as such, can only visualize behavioural models

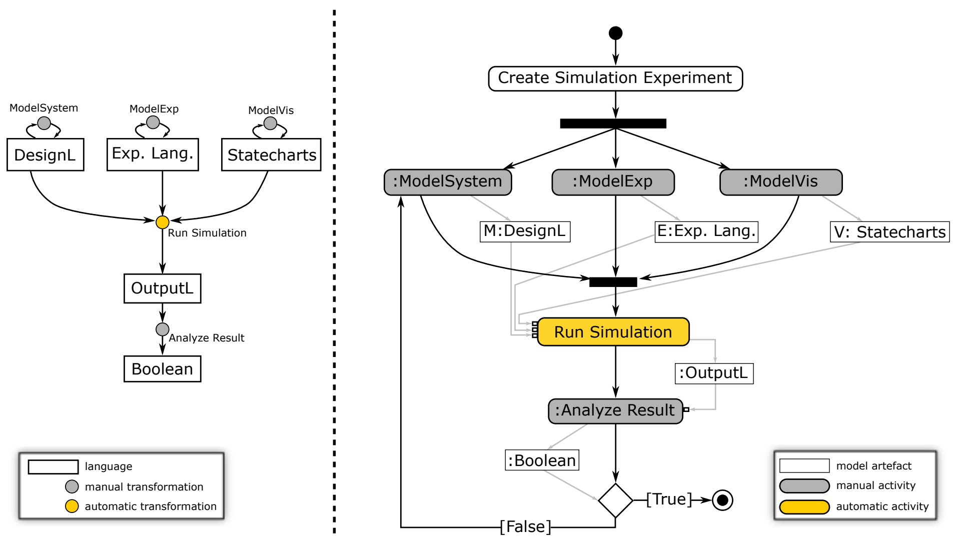


Figure 2.9: A modelling and simulation workflow modelled in a FTG+PM language.

with a communication mechanism). If no visualization is required, the simulation can just as well be run without it ("headless").

Once these artefacts are created, the domain expert runs the simulation experiments and observes the results (typically, a trace generated by the simulation model and/or the visualization). The domain expert then analyses the results and decides whether the goals of the experiment are satisfied by the model. If so, the system is adequately modelled. If not, the modeller has to adapt the model.

This workflow emphasises modularity, as it splits the simulation system into separate subsystems. It allows the domain expert to replace components as necessary, and increases scalability, as components can be deployed on separate machines. In many cases, a process such as this can be employed by domain experts to design and evaluate their systems (for example, to simulate occupant behaviour in buildings [22]).

In this thesis, we assume modellers use a workflow similar to this one to model their systems.

Summary

This chapter provides the necessary background for the techniques developed in this thesis. We started by motivating the MDE and the MPM philosophy for developing complex engineered systems. We then explained language engineering techniques that allow to build tools for the many different languages used in a typical MDE workflow. These techniques allow to model a language's syntax (using metamodelling), its semantics (often using model transformations) and an engineering workflow that can be enacted for guiding the engineer(s) during system development. They have effectively been used to generate modelling environments and accompanying tooling such as simulators and verification and validation tools. We've explained how complex timed, reactive, autonomous systems can be developed using the Statecharts language. Last, we tied these concepts together into a typical modelling and simulation workflow, in which modellers use the available tools and languages to model, simulate, and analyse their systems. This scopes our work, since we contribute language engineering techniques for building model debugging environments. Before we present these techniques, the next section surveys the state of the art in debugging techniques, both for programming languages (well-established) and modelling languages (more recent).

Chapter 3

State of the Art

This chapter presents the state of the art in debugging that is relevant for this thesis. It is divided into a number of sections, each listing a set of techniques, methods, and tools for debugging a subset of systems. In Section 3.1, the large body of work on program debugging is explored. We go into a number of subtopics, including techniques for debugging concurrent programs, omniscient debugging, and live programming. This section provides a solid foundation for Section 3.2 on model debugging, where we hone in on the topic of this thesis and discuss what has already been done. Since models are abstractions of real-word systems or systems implemented using software, their debugging techniques differ significantly. The section surveys the growing research body on model debugging, providing an overview of these debugging techniques. Section 3.3 touches on the topic of simulation debugging. Since simulations are not necessarily implemented explicitly using models, but are significantly different from traditional software systems (due to the semantics of the simulation abstractions used), other debugging methods are needed. Last, we survey the back-translation of traces obtained from simulation, execution, or verification to a format using abstractions familiar to the modeller of the system.

3.1 Program Debugging

Developers of software systems spend a large portion of their time debugging the code they write, incurring a high cost [145]. Many approaches exist to minimize the amount of defects developers introduce in their code; as systems become increasingly complex, however, no method guarantees failure-freeness. Such approaches include early system verification and testing, but debugging remains an important activity [71]. Whenever a new development paradigm, programming language, or development environment is introduced, debugging techniques for that platform follow shortly after. And, while program debugging is well-researched, code is still the most popular vehicle with which to produce (software) systems. As such, it is ever-evolving. This section surveys well-established program debugging techniques, as well as more recent ones.

In Section 3.1.1, we look into techniques for debugging traditional, sequential programs. In Section 3.1.2, we review techniques for debugging concurrent programs, which can be run on a number of (logical) parallel processes or distributed over multiple (physical) processors. In Section 3.1.4, we explore “omniscient debugging”, a technique for traversing the execution trace of a program both forwards and backwards. In Section 3.1.5, we survey techniques for “live programming”, a more recent technique that allows to modify the code of a program while it is executing.

3.1.1 Scientific Debugging

This subsection explains techniques for debugging sequential programs, which have been well-researched and serve as a basis for the techniques explained in the next sections. The field of program debugging is well-established and many works provide guides to using them effectively. Zeller [223] provides an excellent overview of existing code debugging techniques. His book explains how to combine these techniques in a process called “scientific debugging” to find, manage, and remove defects in software in an optimal way. The first step when observing a failure, is to reproduce the problem by reproducing the environment and conditions that lead the program to fail. Next, a number of iterations have to be gone through where the programmer invents a hypothesis (or refines a previous one), predicts the program behaviour, and tests the hypothesis by experimenting with the program, leading to a rejection or a refinement of the hypothesis. Ultimately, a hypothesis can no longer be refined, and the programmer arrives at a diagnosis for the failure: the defect has been found.

This subsection surveys commonly used debugging techniques that aid the programmer in the experimentation phase of the scientific debugging method. The main task of any debugger is to convey the meaning (runtime semantics) of the code to the user, explaining why some behaviour is observed. Debugging techniques can provide functionality to the user to manually traverse the runtime behaviour of a program, relying on the user to control execution and observe state changes, or they can be automated, guiding the user towards the source of the defect (semi-)automatically.

Most, if not all, programmers are familiar with interactive debuggers that provide functions to pause, resume, and step through a program. Such debuggers also offer functions to observe the state of the program (the call stack and variable values). Most debuggers reuse the symbols introduced in the program text to present the information to the user with a minimal semantic gap.

A well-known open-source debugger is the GNU debugger (gdb) [62]. It was developed for the Linux operating system and the C programming language, but has been turned into a multi-platform, multi-language debugger by separating the symbol side (where the source files of the program are found) and the target side (the platform on which the compiled binary runs). This separation allows gdb to debug native binaries, remote programs, and stored traces through a common interface on the target side: the target vector. For the Linux platform, this vector is implemented using the *ptrace* tool, which allows a process to attach itself to another process and control it as well as read its runtime state. On the symbol side, a symbol table translates memory locations to variable and function names, although gdb can handle partial or missing symbol tables, in which case raw memory addresses and values are presented to the user.

Lower-level debugging can be provided by Dynamic-Binary Instruction (DBI) systems such as Pin [121] and Valgrind [146]. These systems allow a user to instrument the binary with instructions to perform an action at strategic locations, for example a memory write or a function call. Many tools, such as profilers, memory checkers, and debuggers can be developed using DBI. Their main feature consists of dynamically instrumenting a running binary, minimizing the overhead for the user.

Although traditional debuggers such as gdb have evolved, it remains a debugger for sequential, procedural languages. Many languages, however, regard other entities as first-class citizens. Most notably, the development process using object-oriented languages revolves around objects, not statements or functions. A debugger that was built for a procedural language might not be well-suited if it assumes other abstractions than the ones the user develops the program with. Object-centric debugging [167] augments traditional debuggers with operations that access or modify objects. A higher level of abstraction is also provided by event-based debugging [149], where users can specify high-level events based on lower-level events such as variable accesses and method calls. These debuggers provide a debugging language based on a dataflow approach to specify events. It allows to monitor the execution of a program at the level of the user's abstractions, encoded in events. Similarly, Marceau et al. construct a dataflow language for scriptable debugging [124]. Scripts can monitor the runtime behaviour of a program and generate high-level events.

Visualization of the runtime state can help the user to understand the program by displaying the data manipulated in a more intuitive way. The Data Display Debugger (DDD) [224] is a graphical debugger that allows to display data structures made up of records that have pointers to each other. A more advanced system is presented by Cross et al. and provides custom visualizations for data structures [44] such as linked lists, stacks, and hash tables. xDIVA allows to create visualization metaphors for data structures, which are composed of other (more basic) metaphors [36]. For visualizing call paths, the Code Bubbles paradigm was implemented in the Debugger Canvas [21].

Query-based object-oriented debugging allow users to write queries on the runtime state of the system, which is a graph of objects [110, 111, 112, 160]. The queries can check relations between objects and their attribute values. Closely related is the Program Query Language (PQL) [125], which is used to dynamically find occurrences of a pattern in the state trace, signifying an error. This can be useful to check for common errors such as the misuse of APIs. Expositor [95] treats state traces as first-class-citizens and allows its users to query, filter, and ask questions of a program's trace (which is a list of program state snapshots). While powerful and highly customizable, the users are required to write debug scripts instead of using the more traditional debugging operations.

Debuggers can provide a higher level of automation. A recent survey by Wong et al. provide a catalogue of fault localization techniques [215], demonstrating its importance for efficiently finding the source of a failure. We cover a number of such techniques in the following paragraphs.

Slicing techniques allow a user to discover which parts of the code of a program can influence a set of variables at a location in the program (the slicing criterion). A study on program slicing techniques is presented in [184]. Slicing has been used extensively to, in general, discover program elements that can influence a location of interest. Or, they can be used to discover those program elements that are influenced by a location of interest. It helps the user in the scientific debugging process to more quickly find the source of an erroneous computation.

Algorithmic debugging automates the debugging process further by attempting to eliminate working parts of the code as sources of observable failures. In [183], Silva surveys the state of the art in algorithmic debugging. In essence, an algorithmic debugger splits the execution of a program which leads to the observation of a failure in sub-computations. It then asks an oracle (most of the times the user), whether the result of a sub-computation was correct or not. it can then progressively ‘zero in’ on the offending statement. Related, Whyline [98] allows users to ask why something was (or was not) found in the program state. The system attempts to find an answer (or multiple answers) to the question, which guides the user progressively towards the defect by continuously asking questions.

Iterative delta debugging [4] assumes that there is an earlier version of the program which does not have the observed failure. By successively going back to older versions, the system finds one which does not exhibit the failure and computes the minimal difference between the earlier and current version of the program, arriving at the change that introduced the defect. Related to iterative delta debugging is angelic debugging, which automatically identifies expressions in a buggy program as ‘repair candidates’ if they can be changed such that a failing test passes and no passing tests break. [34]

In model-based software debugging, a model is built that reflects the behaviour of the (incorrect) program, while test cases specify the correct (or expected) result. When a inconsistency between expected and actual results is observed, a set of model elements is computed that, when they behave differently, explain the inconsistency. A model-based debugging engine computes possible explanations for the inconsistency. An overview of such techniques is given by Mayer and Stumptner [126, 127].

Although many automated software debugging techniques exist, they are often not used in industry, where professionals resort to the breakpoint-based debuggers and even print statements [182].

3.1.2 Debugging Concurrent Programs

Concurrent programming allows a developer to split sub-computations of a program into concurrently executing threads of control. It improves modularity, and, in many cases, performance, especially if these threads of control are deployed to parallel processes or distributed over multiple physical processors. Naturally, concurrent systems increase the complexity of systems since the threads of control can exchange information, and need to be synchronized at certain points in time; this can lead to subtle defects such as race conditions, deadlock, and starvation. This is further complicated by their inherent non-determinism: the threads are independently scheduled and can be preempted at arbitrary points in their execution. Observing a failure can be difficult or impossible of two executions of the system with the same inputs exhibit different behaviour. And, when “probing” the system, another effect can appear: the “Heisenbug”, which disappears (or appears) when the behaviour of the system is observed (since instrumenting concurrent programs can lead to a different scheduling of the threads).

McDowell and Helmbold survey techniques for debugging concurrent programs [131]. They distinguish four categories of approaches: traditional (breakpoint-based) debugging techniques, event-based debugging, techniques for displaying flow of control and distributed data, and static analysis.

Several traditional, sequential debuggers may be coordinated by a “master” debugger: each thread of control has an attached debugger that allows the user to set breakpoints, step through the code, and observe the state. The information has to be presented suitably: often multiple windows (one for each thread) is shown. We highlight a number of works that implement this approach. Kacsuk et al. present an approach to debug message-passing parallel programs based on collective breakpoints and macrosteps [89]. Collective breakpoints pause execution when messages are exchanged by the parallel processes. A macrostep is the code executed between two collective breakpoints. These high-level breakpoints and steps provide new tools to developers of message-passing parallel programs, closer to the information they require: the (often problematic) communication between processes. Balle et al. aggregate single-process debuggers for parallel programs [7]. They aggregate the output from lower-level debuggers to present the information to the user in a manageable form. Users can focus on individual processes and debug them using a traditional code debugger. Cunha et al. develop a generic framework for parallel and distributed debugging that coordinate a set of single-process debuggers [45]. Leske et al. [113] improve the information on the running threads provided to the user by constructing full thread histories. These histories allow a user to see a merged view of the thread’s stack and its parent’s stack as if it were a sequential program.

Bates develops a debugger for distributed systems using event-based models [13]. They see a system execution as a stream of events that signal the occurrence of significant behaviour. By building a mode of expected system behaviour, the generated event trace of the system execution can be matched to its specification. If the actual behaviour does not fit the user-defined patterns, a potential defect is found. Similarly, Fromentin et al. present an overview of replay mechanisms and detection of properties of distributed executions [56]. They assume message-passing distributed systems with no global clock. Each process maintains a history of events, over which a user can abstract to construct high-level traces. These event traces are then analysed, to detect concurrency bugs such as racing.

Kraemer and Stasko survey visualization techniques for parallel systems [101]. They present systems that visualize properties of the system that users are interested in. These properties range from communication patterns between processes, temporal ordering of computations, statistical displays for performance evaluation results, memory access displays, barscope views to show the mapping of threads to processes, and application-specific displays that allow for the user to customize what information is displayed, and in which form. Pancake focuses on the way information on parallel system execution is presented graphically to the user [150, 151]. The main requirement of graphical displays is to make quantitative information intelligible, essential for parallel debuggers which need to process a large amount of data. Koch and Zündorf use UML object diagrams to visualize the state of distributed applications at runtime [100]. JaVis visualizes concurrent Java programs using UML diagrams [132]. Caerts et al. [31] implement a hierarchical graphical debugger for communicating processes. They propose the use of domain-specific visualizations to visualize hierarchical processes, which allows users to employ a top-down approach, successively zooming into processes of interest.

Since many defects in concurrent programs stem from timing-related behaviour, temporal properties are an important tool for users wishing to verify the behaviour of these systems.

A flexible approach to high-level stepping is presented by Gunter and Peled, who allow the user to define steps using temporal logic [70]. The program and step definitions are translated to finite state automata. Their approach is not as powerful as traditional model checking, but has a lower cost and allows users to explore the semantics of the program using (temporal) properties they are interested in.

To deal with the non-determinism of concurrent programs, debugging techniques are developed that approach the issue from two directions: either they allow to replay the process deterministically (using record and replay techniques), or they embrace the non-determinism and allow the user to explore alternative branches in the state space. A review of replay techniques is presented by Ronsse et al. [170]. The main challenges for the record-replay technique for concurrent problems are the overhead for the recording phase (to avoid introducing or removing bugs) and faithful re-execution, which can be attained with data-based replay (forcing the processes to read the same values as during the traced execution) or ordering-based replay (forcing the order in which processes interact, which requires a logical clock that partially orders process interactions during tracing). Wang et al. also present a survey of deterministic replay mechanisms for multithreaded debugging [211], distinguishing techniques based on their tracing method (hardware-based, software-based, virtual-machine based, or hybrid). Jockey [173] and BugNet [144] are two tools for recording and replaying the execution of a program by recording calls to non-deterministic functions such as input/output. For concurrency-related bugs such as deadlock, Zamfir and Candea introduce a method for automatically synthesizing a thread schedule and program input for the failure to manifest itself [220]. The synthesized execution can be replayed deterministically using a traditional debugger. Kobler et al. embrace the non-determinism of distributed programs. They allow users to explore other branches in the non-deterministic execution of OpenMP programs resulting from date races and synchronization races [99]. If users find potential race candidates, they can re-execute the program and force a different choice of event ordering (process communication).

Last, we explore debugging techniques for multi-agent systems. Van Liedekerke and Avouris explore the debugging of multi-agent systems and provide a technique based on an abstracted view of the execution [115]. They stress the importance of presenting the state of the system in a way that is natural, using abstractions related to agents. They develop an interface which visualizes both the local view of agents, as well as their communication patterns. Poutakidis et al. [164] present related techniques. They assume complex multi-agent systems are developed using high-level languages such as UML. From such design artefacts, a test set can be produced which checks whether the implemented system conforms to the specification documents. Conversely, the behaviour of the system can be observed and presented to the user using the abstractions of the modelling language(s) used to design the system. A more advanced approach is presented by Hindriks [82]. The system he presents allows users to ask questions about the system behaviour: the debugger can explain why an action was (not) performed, and why an agent does (not) have a certain belief or goal.

From this overview, we can conclude that many systems attempt to hide the low-level behaviour of the system and present a global overview of system behaviour that is more easily interpreted by the user. This is offered through high-level events detected by patterns on the execution trace and high-level steps through the system. Trace information is visualized in a format a user is familiar with; in some cases, design artefacts (models) are reused.

3.1.3 Debugging Embedded Systems

Embedded systems are often characterized by hard real-time deadlines, they often control hardware systems, and they need to interact with the environment. This increases their complexity compared to traditional, often isolated, pure software systems.

Hopkins and McDonald-Maier survey the debug support for complex systems-on-chip [85], where they list four requirements for effective debugging support: 1) there should be limited change in device behaviour; 2) users should be able to externally observe the system state and other critical nodes; 3) users should have external access to control the system state and resources; 4) the cost impact should be limited in terms of device pins or chip area. Pouget et al. [163] develop a debugging technique for dataflow applications. The debugger presents debugging information (such as the state of the system, data dependencies, and call graphs) using dataflow abstractions. Burgess et al. [29] develop a tool that can instrument embedded systems while they are running to observe their behaviour and debug them at a high abstraction level (through event monitoring). Tsai et al. [195] introduce a technique for monitoring the execution of a target (real-time) system non-intrusively, and later to replay that captured trace deterministically. Similarly, Thane et al. develop a deterministic replay approach for real-time systems using standard components [194].

3.1.4 Omniscient Debugging

Omniscient debugging allows users not only to explore the state trace of a program in the forwards direction (by stepping over statements), but also backwards. This is motivated by the observation that it easier to find the location in the code where a failure is observed (for example, an exception is thrown, or an incorrect value is detected). With traditional code debuggers, users need to restart the program multiple times and manually track back from the failure to the defect. Omniscient debuggers allow users to step back in the execution trace, eliminating the need to restart the program and manually track back. The main issue with omniscient debuggers is memory consumption: maintaining a full execution trace quickly becomes unmanageable.

Engblom presents an overview of different techniques for omniscient debugging of GPLs [50]. Recently, support for omniscient debugging has also been included in mainstream tools, such as the gdb [61]. For the object-oriented paradigm, a back-in-time debugger was created by Lienhard et al. [116].

Pothier et al. use events to monitor the running execution [162]. These events are stored in a database, which can be distributed to further increase performance. They also support partial traces by selectively generating events to decrease storage requirements.

Boothe explores techniques for efficient bidirectional debugging of GPL programs [17]. These techniques are based on event traces (to deal with the non-determinism) and snap-shotting (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 [222], partly avoid the problem of memory consumption. But while they avoid one problem, reverse computation is computationally more intensive for long jumps, and not always possible.

3.1.5 Live Programming

Live programming is a more recent technique which allows users to change the code of a running program to quickly test hypotheses. Systems implementing live programming need mechanisms for replacing the executing binary with the new version, as well as to merge the runtime state of the old program with that of the new version of the program.

One of the most important distinctions between different approaches is how they handle time: a distinction is made between *real-time* and *recorded* [130]. 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 [174], Smalltalk [64], Erlang [3], and SELF [198]. In recorded mode, all past input events are recorded and replayed, resulting in a completely new history. This is implemented in languages such as ElmScript [46] and YinYang [130].

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 [49, 69, 129, 155]. Many different types of languages have been made live: graphical languages such as VIVA [193] and Flogo [72], textual languages such as ElmScript [46] and Smalltalk [64], and hybrid languages such as Subtext [49].

Many challenges related to live modelling are tackled only for specific cases or specific languages. One issue relates to how the state needs to be retained [53, 187], and what needs to be recomputed [30]. Making an existing programming language live is often done through ad-hoc modifications, often turning liveness into a black art [28].

3.2 Model Debugging

Models are used to describe systems on a higher level of abstraction by focusing on *what* the system does, not necessarily *how* it does it. This allows users to focus on the essential complexity of the system, instead of implementation details. Model-driven methods, by providing abstractions with which modellers are familiar with, attempt to decrease the amount of implementation errors made. That does not mean, however, that the models created are without defect. Failures can still be observed, and debugging techniques are necessary to trace their source. Debugging models is considered important, for example in the automotive domain [188].

In Section 3.2.1 we explore how executable models can be debugged. In Section 3.2.2 we look at a specific subset of languages: domain-specific languages, whose semantics are often defined by mapping onto a programming language, and how debuggers for these languages are constructed. In Section 3.2.3, we shortly discuss non-executable modelling, for which debugging techniques have also been introduced.

3.2.1 Executable Modelling

A plethora of executable modelling languages have been developed. In one way or the other, they model an executable process, that can be executed through simulation, interpretation, or compilation. With the design and implementation of a new language comes the need to develop tools that aid users of the language to construct, execute and debug models in the language. In [2], Allen et al. explore requirements for modelling and simulation tools that support verification, validation and testing. One of those is the need to present concepts at the level of the concepts of the modelling language. This section discusses the work by the research community on enabling debugging for executable modelling languages—some of the debugging operations offered are transpositions of code debugging operations, others are language-dependent.

One way of implementing debugging is through instrumentation. Since the model is executable, it can be instrumented in such a way that user interaction is added that implements debugging operations such as pause/resume, stepping, and breakpoints. An example of this approach was presented by Mustafiz and Vangheluwe [142]. They generate an experimentation environment for Statecharts models by "embedding" the original Statecharts model into the Statecharts model of the environment, which implements the user interaction for debugging. The instrumentation is implemented using explicit model transformation. The model is further instrumented to visualize the current state in the original model. The instrumented model runs in the background, while the user interacts with the original model in the experimentation environment. There is support for execution modes, steps, breakpoints (by specifying a condition), and changing of the scale factor in real-time simulation. Bagherzadeh et al. [6] similarly instrument a UML-RT model (which shares some commonality with the Statecharts language) with debugging operations.

Slicing Statecharts models was implemented by Luangsodsai and Fox [118]. They distinguish two types of dependencies in non-concurrent Statecharts: control dependence, where the entering of a state depends upon the previous state, the triggered event, and the condition; and data dependency, where a test on a variable depends on that variable's definition. Three more dependencies are introduced for concurrent Statecharts: parallel control dependence (parallel components are entered simultaneously), interference control (an event is raised in a parallel component, causing a transition in another parallel component), and interference data dependence (a variable is referenced in a component, but defined in another). The different dependencies are represented in an and/or-graph. A slicing criterion can be defined, based on a state or event (in combination with a state). The Statecharts model is then sliced (backward or forward), so only relevant states and transitions remain.

Model debugging has received some attention in the literature on the Modelica language [55], which is an a-causal language for modelling physical systems using equations. In [5, 26, 27, 157, 158, 159, 185, 186], the authors develop techniques for debugging equation-based models, which differ greatly from sequential programs, where each statement is executed sequentially. They look at static and dynamic debugging, as well as how to make the debugging techniques scalable for large models. Further, they develop algorithmic techniques that use slicing to (semi-)automatically find potential source of failures. Their dynamic debugger (offering breakpoints and stepping) is implemented by instrumenting the generated code with symbolic information and translating high-level debugging commands to low-level C debugging commands.

Tools for Petrinets, an language that allows to model non-deterministic behaviour, offer limited debugging capabilities. An example is TINA [14], which allows to step through a model's firing sequence manually, but does not offer more advanced capabilities such as breakpoints.

SpecDiff uses model differencing to aid users in understanding the evolving behaviour of formal specifications [217]. They construct, for both versions, a model to capture the operational semantics of both models. These models are compared, and the differences (points at which the behaviour of the models is different) are presented to the user as possible defects.

An alternative approach to model debugging is presented in [122]. The authors argue that the traditional DEVS [221] terminology does not communicate the meaning of the formalism to domain experts that might not be familiar with discrete-event modelling and simulation. They introduce a new, more intuitive way of specifying discrete-event systems, explicitly exposing the semantics of the DEVS formalism. They also present a novel visualization method that communicates the dynamics of the system (in the form of an event trace) more effectively to the domain expert. While they do not provide debugging operations, the information displayed in their interface can be useful in a debugging context.

Model Transformations (MTs) are an essential part of the Model-Driven Engineering (MDE) approach to designing and developing complex systems. Early work on debugging graph rewrite rules (on which MTs are based) is described for Fujaba by Geiger and Zündorf in [63]. Mészáros et al. [135] implement a debugger for the Visual Modelling and Transformation System (VMTS). They implement step-by-step execution, breakpoints, visualization of the current state of the transformation and host model, and runtime modification of identified matches, the host model, as well as the transformation definition. Similarly, Sun and Gray [190] extend their demonstration-based model transformation engine with debugging support, allowing users to step through previously defined patterns when they are deemed incorrect. Low-level execution information is hidden, and the debugger presents the information graphically in a format that is familiar to the users of the tool. In [176, 177, 178, 213], Schoenboeck et al. make the observation that MTs are inherently non-deterministic. To capture that non-determinism, they develop a domain-specific language on top of Petrinets [139] to execute and debug MTs. Corley, on the other hand, focuses on bringing debugging support similar to program debuggers, in particular omniscient debuggers, to debug MTs. [40, 41, 42, 43]. To achieve this, he transposes "step forward" and "step back" operations from code debugging to appropriate debugging operations that allow traversal of the MT execution history in both directions. Since model transformations are non-deterministic, the implementation logs each change at the end of a transformation step, to be able to undo them when the user requests to step back. By inverting these changes, users step back to previous states. Overhead is limited, as it is incremental in nature, though the debugger eventually runs out of memory unless old events are dropped or persisted to disk. Hibberd et al. [80] implement forensic debugging of MTs: traces of the transformation's execution are leveraged to ask "why (not)" questions. Ujhelyi et al. [197] transpose slicing techniques onto MTs to find potential sources of an observed failure. Dynamic backwards model slices are computed based on data and control dependencies that were computed from the MT definition. More recently, Jukšs et al. [88] describe a “layered” approach to the debugging of MTs, recognizing the embedding of languages (scheduling, rule, pattern). They allow users to specify debugging “scenarios” using model transformations.

Closely related are debugging techniques for Model-Driven Development (MDD), and more specifically for the Unified Modelling Language (UML), a family of (executable) languages for specifying the structure and behaviour of software systems. In [103], Krasnogolowy et al. map code debugging concepts onto the Story Diagrams formalism, and build a visual debugging user interface in the open-source development environment Eclipse. Their architecture consists of a debugging client (with which the user interacts) that communicates with a debugging server, which controls the execution of the interpreter. Both Mayerhöfer [128] and Laurent [107] extend the fUML—a subset of the UML for which precise semantics are defined—with debugging support. Dotan and Kirshin similarly develop a visual debugging tool for behavioural UML models [48]. Pópulo is a debugging tool for UML models [57]. It implements step-by-step execution and breakpoints for UML activity diagrams. Brüning et al. implement debugging support for OCL constraints in UML models [24]. Graf et al. [67, 68] propose debugging embedded systems that were specified using UML models by mapping runtime information back to the model level. Specifically for the state machine language, Kaufmann et al. [90] propose a SAT-based tool that checks the consistency of a state machine with its specification (in the form of a sequence diagram). In case a counter example is found, it is visualized in the graphical modelling environment, which allows the user to step through the trace.

3.2.2 Domain-Specific Modelling

Recent advances in modelling language engineering have provided language engineers with tools to quickly develop a specialized modelling language and its associated tools, including (graphical) modelling environments, simulators, and code generators. DSLs allow an engineer to focus on *what* the solution to a problem is, not *how* it is implemented. The solution is developed in a notation that is most familiar to the engineer. For domain-specific models, focus has primarily been on the theoretical foundations of (meta-)modelling [105] and how (domain-specific) modelling can help developers [91]. Nowadays, focus starts shifting to model execution [133]. With model execution comes the need for debugging, and since the effort to develop domain-specific tools should be as low as possible, techniques for developing domain-specific language debuggers are needed.

Mannadiar and Vangheluwe [123] address the need for debugging models in domain-specific languages. They propose a mapping of code debugging concepts (such as breakpoints) to model-based design. Their focus is on debugging model transformations and synthesized applications (with traceability links back to the design models from which the code was synthesized).

Wu et al. [216] map DSL debugging operations to code debugging operations, performed on the generated code. The language developer has to instrument the grammar of the (textual) DSL with debugging actions. Similarly, Lindeman et al. [117] instrument a language definition with debugging support. The debugger specification is expressed in a specialized language called SEL, and the parsed DSL program is instrumented before code generation to insert the appropriate debugging behaviour. The instrumentation specifies points in the code where domain-specific events are generated and communicated to the domain-specific debugger. A debugging environment interacts with the generated code to offer the user a domain-specific view on the execution state and a set of debugging operations (pause/resume, step over/into/out). Their approach is platform- and language-independent, and their architecture is split into four parts to isolate platform-specific libraries. Finally, Sadilek and Wachsmuth describe a method for building (visual) debuggers for DSLs by (manually) instrumenting the operational semantics of the DSL [172]. This allows language developers to quickly prototype language workbenches that implement debugging. TIDE is a generic framework to develop debuggers for (domain-specific) languages [199]. The language developer needs to define a number of debugging rules, that make use of logical breakpoints (DSL-level steps) to implement debugging actions.

Pavletic et al. [153, 154] present debugging techniques for extensible languages. Such languages extend a previously defined (general-purpose) language with extra domain-specific concepts, while still retaining the full power of the underlying language. Chiş generalizes techniques for adding debugging support to DSLs in the Moldable Debugger [37], a reusable framework for developing debuggers for DSLs. The framework allows to implement a set of debugging operations such as stepping, state querying, and visualization at the most appropriate (domain-specific) level of abstraction. The framework is flexible, and determines the most appropriate debugging view depending on patterns in the code.

Bandener et al. [8] demonstrate how to add visual execution and debugging to a DSL whose syntax is described in a metamodel, and its semantics in a set of graph transformation rules. Similarly, in [19], Bousse et al. describe a partly generic debugger that can be extended with domain-specific trace management functions [20]. They allow the definition of a set of debugging operations that traverse, query, and manage these execution traces.

In [200], van der Storm explores how executable (domain-specific) 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.

3.2.3 Non-Executable Modelling

While debugging techniques are most often associated with exploring the execution of a system, debugging techniques have been developed for non-executable modelling languages. These techniques attempt to find defects in the specification of the models, most often by explaining the results of a well-formedness checker.

Wille et al. [212] develop techniques for debugging UML models that do not conform to their metamodel, either violating UML constraints, or violating OCL constraints. If an inconsistency is found, only a small subset of the model’s elements have to be considered by the user in order to re-establish the conformance relation between model and metamodel. Similarly, Beanbag [218] is a language that allows to (semi-)automatically re-establish the consistency relation between a model and its well-formedness rules (which are similar to OCL constraints) after a user has edited the model.

Ananke is a tool for debugging constraint models based on metamodels and metaknowledge [54]. The authors present an iterative process for improving constraint satisfaction models. The process is guided by suggestions for relaxing or tightening the constraints of the model.

3.3 Simulation Debugging

Simulations are a way of answering questions about systems without physically building them. They can be implemented using various techniques, in general-purpose programming languages or specialized simulation languages. Whatever the implementation medium, debugging techniques aid the simulation developer in finding defects.

Krahl presents a "best-practice" guide to simulation debugging based on common types of defects in simulation models, when implemented in a programming language [102]. Among the tools useful for debugging, the author lists animation, model traces, and debuggers for interactively stepping through the simulation code.

Debugging techniques are often based on a view on the trace of the simulation, which possibly aggregates some parts or, through visualization, aids the user in finding patterns that demonstrate a defect is present. In [25], Buchanan and Keefe explain how debugging support was added to the Möbius modelling and simulation framework, which provides a formalism-independent discrete-event simulator. They add support for stepping, model state modification and model state visualization to their existing kernel. The back-end simulation process is separated from the front-end visualization by a communication layer. Kemper [92] presents a method for debugging stochastic simulation models based on a visualization of their trace, from which irregular patterns can be discerned. In [66], Gore et al. develops a different approach, extending statistical debugging techniques traditionally used for debugging software systems, which makes them effective to debug simulation systems as well. The author assumes the models are developed using a programming language, instead of a high-level modelling language. By introducing simulation-specific debugging aids, traditional code debuggers are extended to help a simulation developer more quickly identify the source of a failure.

Rogin and Drechsler build a high-level debugger for the SystemC simulation language [168]. The debugger is based on gdb. It abstracts its output to offer system-level information to the user of the debugger. Operations are also provided at the SystemC level and translated to commands for gdb.

In [93], Kemper and Tepper present an automatic transformation from simulation models to Petrinets models, in which analysis can be performed. The analysis results can discover failures in the simulation models and point to their source.

3.4 Back-Translation of Traces

When using specialized modelling or simulation languages, models are often translated to another language (the semantic domain) for simulation or analysis. This approach promotes reuse of languages whose semantics are well-defined and whose tooling is mature. There can be a wide semantic gap between both languages, however. Users of the source language might not be familiar with the abstractions used by the target language. Back-translation of simulation or verification traces is useful, as they present the trace using abstractions most familiar to users of the language.

As part of the ProMoBox approach, verification results (counter examples) are translated back to a domain-specific trace language [136]. The model is a trace leading to a violation of a property, and can be stepped through in a visual environment. The back-translation is performed by rules that are derived from the modelling language's semantics.

Related to the ProMoBox approach is the approach presented by Zalila et al. [219]. They instrument the operational semantics of the modelling language (expressed using model transformations) using higher-order transformations to produce traceability information. That information is used to translate verification results back to the domain-specific level.

Hegedüs et al. [79] use a back-annotation technique to translate simulation traces in a target formal verification tool (and accompanying language) back to domain-specific concepts. The approach presented in their paper is used to translate the output model in the target output language back to an output model in the source output language, making use of the (forward) traceability links between the source and target design models. The back-annotation phase uses *change-driven transformation rules* to find patters in the target state trace that correspond to changes in the domain-specific state.

These techniques improve model understanding, but are not necessarily a debugging technique; after detecting a failure in the domain-specific trace obtained by back-translation, the defect in the model still has to be found.

Summary

Program debugging techniques have been well-researched, and a plethora of debugging tools have been developed. They all have the same goal: program understanding and the tracking of defects in source code. The techniques range from manual step-wise debugging (forwards and backwards), trace visualization and replay, algorithmic debugging, query-based debugging, and model-based techniques. We observe that abstraction is often used to understand increasingly complex systems such as parallel and distributed programs. The sheer complexity of the code makes abstraction essential to navigate the state and execution space successfully. These techniques naturally evolve towards (domain-specific) modelling languages where abstraction is promoted and domain abstractions are promoted to first-class citizens. The first steps to bring debugging to these modelling languages relied on manually implementing them. More recently, efforts to structure the building of advanced (visual) modelling, execution and simulation environments for modelling languages have expanded to include debugging. These methods hand tools to language designers for quickly building a complete set of tools for their language—from editors, to executors, analysers, testers and debuggers. In the next chapter, we contribute to this effort by extending the available methods. We analyse the semantic properties that modelling languages can exhibit, and explore debugging for each semantic variation point. These debuggers are created according to a structured, repeated process and integrated with existing modelling environments instrumentation.

Chapter 4

Modelling Model Debugging Environments

This chapter explains how model debugging environments can be explicitly modelled. We extend the state-of-the-art in (visual) environment construction for modelling languages based on language engineering.

We explained in Chapter 2 how MDE techniques can help overcome the complexity of developing engineered systems. In particular, the multi-paradigm philosophy takes advantage of a wide variety of modelling languages, each with their specialized syntax and semantics (sometimes tailored to a specific domain), to model the different aspects of a system at the most appropriate level of abstraction. But, tool support is crucial, and techniques are required to efficiently build modelling tools that support each phase in the MDE process. This includes specifying, verifying, testing, and simulating models, amongst others. In the previous chapter, we surveyed the state-of-the-art in program debugging and the growing field of model debugging. From the discussion there we deduce that currently lacking is an approach based on language engineering that allows the construction of debugging and experimentation environments for formalisms with varying syntax and semantics. This chapter fills that gap: we provide a structured approach towards building such environments, by providing a workflow, architecture, and an instrumentation technique for adding debugging support to model simulators.

We consider the ProMoBox approach [136] to be the state-of-the-art in language engineering for modelling, simulation, and verification environments. In Section 2.2.3, we explained that, inspired by the ProMoBox approach, we consider in this thesis that a modelling language actually consists of three sub-languages:

- the *design language*, its syntax described by metamodel MM_D, supports the modelling of designs in the language;
- the *runtime language*, its syntax described by metamodel MM_R, supports the representation of a *runtime state* (snapshot) of models in the language;
- the *output language*, its syntax described by metamodel MM_O, supports the repre-

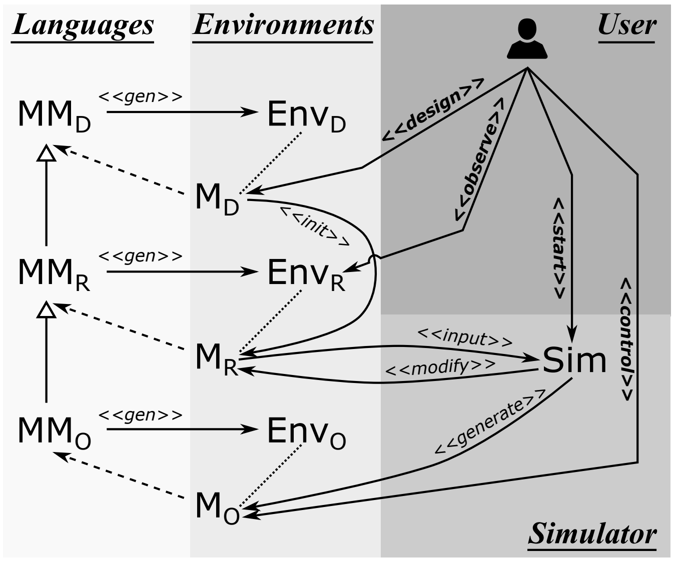


Figure 4.1: The different environments, their relations, and operations.

sentation of an *execution trace* (of consecutive snapshots in the language MM_R) of models in the language, resulting from a simulation or verification run.

These languages are used to generate three environments: a *design environment*, which offers tools to engineers for building designs in the language; a *simulation environment*, which interprets the results from a(n) (external) simulator to represent the evolving state of a model; and a *replay environment*, which interprets an execution trace or verification results and allows a user to step through this trace.

Figure 4.1 presents the three environments (one for each sub-language), their relations and operations. From the user's perspective, these environments allow to perform the following modelling and simulation tasks:

1. In the design environment, a valid design of the system can be created.

2. In the runtime environment, the runtime state of the system can be displayed. The runtime model for a system is obtained from its design model, and initialized according to a user-specified configuration. A runtime state can be used as input for a simulator. The simulator updates the state of the system and modifies the runtime model to display the new runtime state. The runtime environment provides visualization support for the simulation process.

3. In the output environment, a complete simulation trace can be visualized. This trace is generated by the simulator at the end of simulation, and it is used to replay the simulation step-by-step (controlled by the user).

The output environment can already be used for a posteriori debugging of the system. It only offers a limited number of features (visualization of the trace, stepping), and in practice might not be usable since the trace of a simulation can be very large and therefore either not representable in memory, or it might be difficult to find a state of interest. The runtime environment similarly can be used to observe the semantics of a model, since it displays the runtime model while it is being modified by the simulator. Not much control is offered, however, besides the visualization aspect.

The ProMoBox approach distinguishes two more languages:

- the *input language*, described by its metamodel MM_I, which supports the modelling of a trace of input events to model the behaviour of an environment;
- the *property language*, described by its metamodel MM_P, which supports the modelling of temporal properties using domain-specific abstractions.

These languages can support running *experiments* on the models created in the design language, similarly to what was proposed in the experimental frame approach [221]. The experiment consists of an input model (conforming to the input language); the simulator then produces an output model (conforming to the output language) and a model checker ensures this output model satisfies the properties defined in the property model (conforming to the property language).

In the rest of the discussion, we do not consider the input, output, and property languages, since we are concerned only with the interactive debugging of model simulations. It is our goal to extend the existing infrastructure for debugging. The techniques that we develop for constructing model debugging environments need to be general: they need to be applicable to any type of language a language engineer would want to build such environments for. This means three things:

- we need to instrument the language's simulator with debugging support;
- we need to construct a debugging environment that can interact with the debugging-enhanced simulator by invoking operations and interpreting their results;
- we need to leverage any other artefact created during the modelling and simulation workflow if they can serve as a debugging aid.

The sections in this chapter describe our techniques for achieving these goals.

Structure In Section 4.1, we explain the (combination of) semantic properties a language can have; for each valid combination of semantic properties, our techniques need to be able to construct a debugging environment. This is demonstrated in Chapter 5, where we use our approach to build debugging environments for languages that exhibit widely differing semantics. In Section 4.2, we explore debugging operations that are useful for modelling languages. We both look at how code debugging operations can be transposed, and which additional operations are useful in the context of modelling languages. In Section 4.3, we explain our technique of de- and reconstructing a model simulator to instrument it with debugging operations. In Section 4.4, we place the de- and reconstructed simulator in an architecture which couples the components useful for model debugging in an architecture. Last, in Section 4.5, we present the workflow for developing debugging environments, tying together the previous sections.

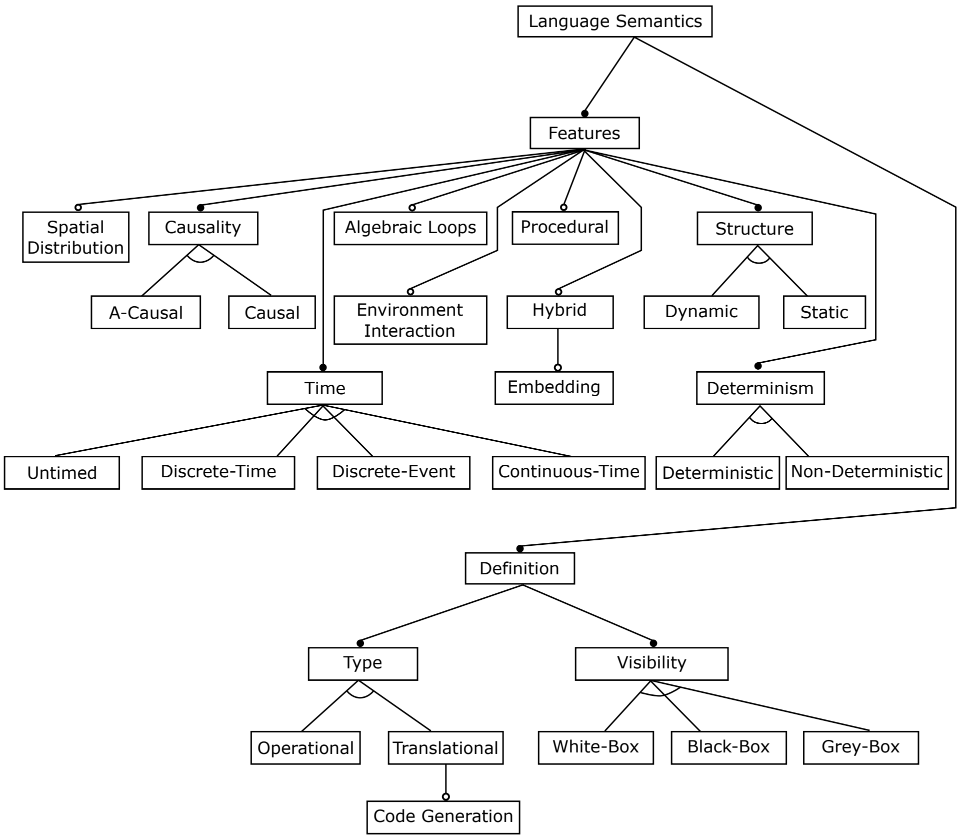


Figure 4.2: A classification of language features.

4.1 A Language Classification

We classify modelling languages according to their semantic properties. This is a useful exercise, since many semantic variations exist among modelling languages. In fact, this diversity in semantics is essential in the context of MDE and MPM, since the domain knowledge of experts is leveraged. We discuss two aspects relevant to a language's semantics for the building of debugging environments: the language's features, and the way these semantics are defined.

4.1.1 Semantic Features

To ensure our techniques are general, we need to consider the semantic variety found in modelling languages. As these languages are used to model systems in a broad context, we are no longer constrained by the sequential, procedural style of programming that still dominates in software engineering. Since most debugging techniques are implemented for software systems, they specifically target such semantics. To transpose traditional debugging techniques onto modelling languages, or create new, simulation-specific ones, we need to study the semantic properties of modelling languages in general.

In this section, we pick out a number of interesting semantic variation points from the point of debugging. Languages can be classified according to the semantic features they exhibit. Often, a language combines several features. Our work focuses on executable languages, and we do not consider languages that have no dynamic semantics. The feature diagram shown in Figure 4.2 presents a classification of modelling languages according to 1) their semantic features and 2) how their semantics are defined. The diagram is a result of examining the semantic properties of modelling and simulation languages that are used to describe the behaviour of systems. In the next subsections, we explore each feature and link them to possible debugging operations. We mention for each semantic property which debugging operations are useful to debug models in those languages. The classification is used in Chapter 5 when we construct debugging environments for a set of different languages: for each language, we specify the semantic properties it exhibits.

Procedures

The structure of a procedural language is decomposed into functions, and a notion of one function "calling" another function is present in the semantics of such languages. Most notably, programming languages traditionally decompose into functions. Even though currently the object-oriented paradigm is arguably most popular (which decomposes programs into objects), the dynamics of the system is still built around calls to the methods of objects. A characterizing runtime feature of such languages is a "call stack" to which an entry is pushed every time a function is called, and one is popped every time a function returns. This gives rise to a call hierarchy at runtime that is built up and torn down continuously while functions are being called and returned.

Procedures have debugging operations associated with them that allow users to navigate this hierarchy: *step into*, *step return*, and *step over*. These operations are directly linked to the calling and returning of functions.

- The *step into* operation is enabled when the next instruction is a function call. By executing the operation, the scope switches to the called function, and the next instruction is the first instruction of the function.
- The *step return* operation is enabled when the current instruction is executed inside of a function. By executing the operation, the scope switches to the calling function, and the next instruction is the instruction following the call to the function.
- The *step over* operation is enabled when the next instruction is a function call. By executing the operation, the debugger resumes execution until the function returns, and the next instruction is the instruction following the call to the function.

These three operations allow the user to navigate the call stack efficiently.

Algebraic Loops

Languages can allow algebraic loops in their models. An algebraic loop occurs if the result of a computation is also an input to that same computation, either by a direct feedback from

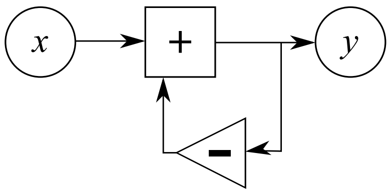


Figure 4.3: An example algebraic loop in a dataflow language.

the computation to itself, or through a number of other computations. Such loops prohibit "normal" computation, since values can depend on themselves. Instead, the dependencies within the loop have to be resolved first and a globally consistent solution has to be found, often by solving a set of equations. A defining feature for such languages is a scheduling mechanism that orders the computation of the new value of the elements based on the analysis of the algebraic loop.

An example is given in Figure 4.3. The value of y depends on the value of x, but also on itself. We construct the set of equations, where w is the output of the negator block:

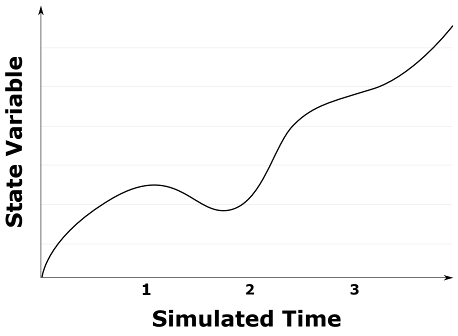
$$y(t) = x(t) + w(t) \tag{4.1}$$
$$w(t) = -y(t) \tag{4.2}$$

While this seems a trivial model to implement in program code, this is not the case: the algebraic loop must be resolved first. Indeed, in code, the statement `y = x - y` translates to the equation $y(t) = x(t-1) - y(t-1)$, as the old values of x and y are used in the assignment. To actually implement the equation $y = x - y$, the algebraic loop must be solved to $y = \frac{x}{2}$, an equation trivially solved in code. But before arriving at that solution, programmers would have to manually solve the set of linear equations to come up with the code to solve the system of equations. In contrast, these formalisms handle linear algebraic loops natively, solving $y = x - y$ automatically. To solve linear algebraic loops, the loop is detected as a strongly connected component, and a linear system of equations is constructed.

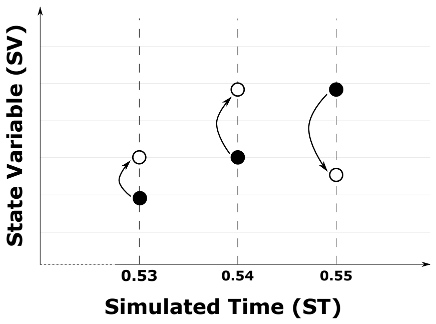
For debugging purposes, languages that allow algebraic loops need to be able to show those loops when they are encountered, as well as the order in which the output of the elements are computed (the schedule).

Causality

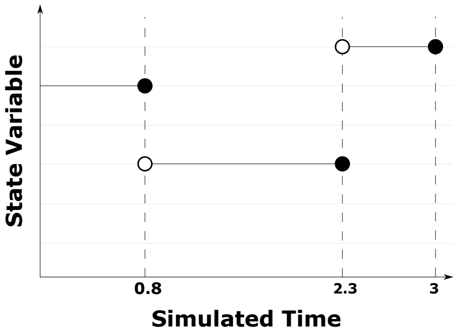
Most simulation languages are causal. On a time-line, events in the present are caused by events in the past and they cause events in the future: they are causally related. Other languages, most notably the Modelica language [55], are a-causal. In those languages, cause and effect cannot be deduced. Instead, relations are defined between model elements, and the execution engine deduces a solution at runtime. It is in some domains more natural to model systems using a-causal languages, since the dynamics of many systems are guided by relations between elements (for example, the relation between current, voltage, and resistance is defined by Ohm's law). Encoding those rules in a causal language would require to translate such relations to (causal) computations, bringing the modeller further



(a) Continuous-time semantics.



(b) Discrete-time semantics.



(c) Discrete-event semantics.

Figure 4.4: Different types of state evolutions as a function of simulated time.

from "what" the system is and closer to "how" the system needs to compute it (for example, in a dataflow language with algebraic loops as presented in the previous section).

As we discussed in Chapter 3, a-causal debugging requires specialized techniques that have been the subject of extensive research. We therefore do not consider such languages for our debugging techniques, and from now on assume the language for which debugging is implemented is causal.

Time

The notion of time is central to many simulation formalisms. The *simulated time* differs from the wall-clock time: it is the internal clock of the simulator. Central to all formalisms discussed in this thesis is a state variable vector (keeping track of the current simulation state), which is updated each time the simulated time is incremented by the simulator. The frequency of these time increments depends on the time semantics of the formalism. We distinguish three categories of timed formalisms, presented in Figure 4.4:

- A *continuous-time* formalism's clock can have any real-number value. The state function changes continuously and is defined for all simulated time instants. To implement such formalisms on a computer, the continuous time function has to be discretized and the state function sampled at equidistant points in time. A continuous-time simulation implemented on a computer can be seen as a discrete-time simulation with a sufficiently small step size. The resulting state function is a stepwise function, approaching a continuous function.

- A *discrete-time* formalism's clock is only defined at equidistant, discrete points in simulated time. The state of the system is updated at those discrete points in time, and is undefined in between two updates.
- A *discrete-event* formalism's clock and state are updated at discrete points in time, but they are not necessarily equidistant. The behaviour of such systems reacts to events from the environment aside from its autonomous behaviour, which can change the state of the system as well.

A last category consists of untimed formalisms, which do not have a notion of time: their simulators do not have an "internal clock". The behaviour modelled by these systems is untimed, although of course, their computation is not instantaneous and has a wall-clock time duration.

A debugger for simulation formalisms has to take into account the time semantics for that formalism. The "stepping behaviour" will be different, depending on when and how often the state is updated. For debugging operations, discrete-event formalisms additionally form an interesting category, since they are reactive to events. These events can be local (*raised* by one of the elements in the model) or can come from the environment. To debug such simulations, the simulation state can be indirectly affected by providing an operation that *injects* such an event at a particular instant in simulated time.

Structure

In a static-structure system, the structure of the system is not part of the runtime state of the system; it is considered static. For example, the model in Figure 4.3 has static structure: its number of blocks and their interconnections cannot change. In dynamic-structure systems, however, the structure of the system can change over time. The formalisms to model such systems allow to natively describe structure-changing behaviour: during simulation, entities can appear and disappear and their interconnections might change. This behaviour is encoded in the elements of the models: they often have the capability to decide that a structure change is necessary, which is performed by a special state change function. This state change does not necessarily affect the element that requested it. An example of a dynamic-structure formalisms is SCCD, presented in Section 2.3.2. Objects implement agent-like behaviour: each object has its state encoded in a set of variable values and its behaviour controlled by a state machine. They can request the runtime to remove or add objects, as well as connections between objects.

Debugging dynamic-structure systems can be challenging. Since entities appear and disappear at runtime, the runtime state of the system is not as easily traced back to its design. The runtime formalism, and consequently, the simulation and debugging environment, differs significantly from the design formalism and environment. In those cases, it might be necessary to look towards other artefacts created for the simulation system. For example, many agent-based simulations are visualized using model-specific visualizations. We discuss such possibilities in Section 4.4.

Determinism

A deterministic system's behaviour is identical in consecutive simulation runs for the same set of inputs (and configuration). A formalism has deterministic semantics if it only allows users to model deterministic systems. In many cases, however, it is possible to model an (unwanted) non-deterministic system if either:

- the semantics of the formalism are not well-defined (leading to implementation-dependent behaviour);
- the semantics of the formalism leave the interleaving of concurrent elements to the runtime system.

Such non-determinism can be the source of many failures in the system and can be notoriously difficult to debug. A non-deterministic formalism, however, includes non-determinism in its execution semantics. At runtime, their semantics branch: the model encodes all possible execution paths, instead of just one. This can be particularly useful for modelling (parts of) systems that are inherently non-deterministic, such as distributed computations, unpredictable environments, and certain scheduling systems.

For deterministic formalisms, we assume the semantics are well-defined and faithfully implemented by the simulator: unwanted non-determinism is ruled out. For non-deterministic formalisms, debugging operations that explore the possible execution paths are necessary.

Spatial Distribution

A system might be spatially distributed. For example, a city consists of a grid of roads, buildings, parks, etc. Many of the properties such elements in the grid posses are very similar, but their composition needs to be considered to be able to analyse the full system. To avoid the tedious task of instantiating the spatially distributed elements in their grid, specialized formalisms allow modellers to quickly define and change the parameters of these models. Together, such networks of "basic" cells can exhibit complex emerging behaviour. An example of a formalism that support spatial distribution natively is Cellular Automata [214]. In computer simulation, CellDEVS [210] has been proposed to model cellular automata. Models in this language are mapped onto traditional discrete-event models, and as such, we do not consider their debugging in this thesis.

Hybrid Behaviour

A hybrid formalism combines two formalisms syntactically and semantically. At runtime, the semantics of the formalisms are interleaved: a coordinator is responsible for the communication between the two formalisms. Semantic adaptation might be necessary to make the execution results of both formalisms compatible. A typical application for hybrid formalisms is the embedding of one formalism in the other: a new syntactic element allows to embed a complete model of one formalism into a model of the other formalism. This creates a parent-child relation between the two formalisms, which at runtime often means that one formalism acts as the "master", while the other acts as the "slave".

In such embedded formalisms, an operation has to be provided that can switch contexts, similar to the *step into* operation of procedural languages. In this case, however, the context switches not from one function to another, but from one language's context to another. As an example, consider the dataflow model shown in Figure 4.3 is a hybrid model, where the function of each block is modelled using an appropriate language (for example, an action language). At the level of the dataflow model as shown in the model, we are interested in its behaviour as a formalism that allows algebraic loops. While executing the model in a debugger, however, we might be interested to look inside the execution of each block. By stepping inside the block's computation, we switch contexts to, for example, a procedural action language.

4.1.2 Definition of Semantics

We've covered the difference between operational and translational semantics in Section 2.2.3. To reiterate, either a simulator is built that can execute models in the language (operational semantics) or a translation is made to a language with known semantics (translational semantics). Operational semantics are implemented by a simulator (or executor). A number of options exist to define the simulator: for example, by encoding its rules in an action language, or by defining model transformation rules that directly modify the runtime model. Translational semantics are most often implemented using a model transformation to another formalism. We consider code generation a special case of translational semantics, where the target language of the translation is a general-purpose programming language. Code generation is often the last step in an engineering workflow where one of the deliverables is a software component. The generated code is subsequently deployed onto hardware and integrated in the rest of the system. Therefore, from a debugging perspective, code generation is linked to deployment rather than simulation.

The user of a formalism is often not concerned with the implementation details of the formalism's semantics. The simulator is, as is the case for closed-source software, delivered as a *black box*, which accepts a valid model as input and produces simulation results as output. If the simulators are vendor-specific, intellectual property protection is another reason for providing the simulator as a black box. In some cases, even the model is black-box: in that case it is coupled with the simulator and offered to the user as a single package with which can be experimented. Debugging in those cases is difficult: if the provided interface does not offer the necessary control, there is no way to instrument the simulator or model with appropriate debugging actions. *White-box* approaches, on the other hand, expose the semantic definition. In those cases, the semantic definition can be altered and instrumented with debugging support. In between black- and white-box approaches are the *grey-box* approaches, where parts of the simulation algorithm are exposed through an interface.

In the remainder of this work, we assume a white-box approach. This is reasonable, as we are targeting language engineers and tool builders, who have access to the semantic definition of a language. In case intellectual property needs to be protected, the developer of the debugger has to construct the debugging interface in such a way as to hide the parts they do not want to expose. These considerations, however, are outside of the scope of our work, and we assume all details can be shared with the users of the debugger.

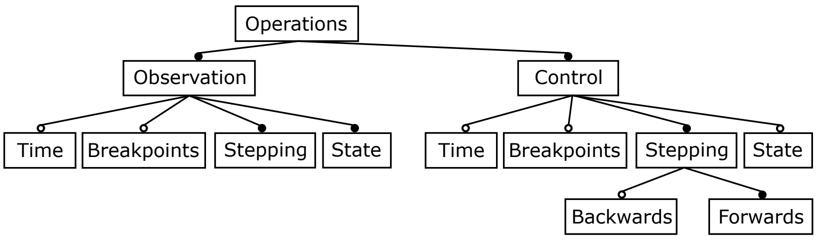


Figure 4.5: A classification of debugging operations.

4.2 Debugging Operations

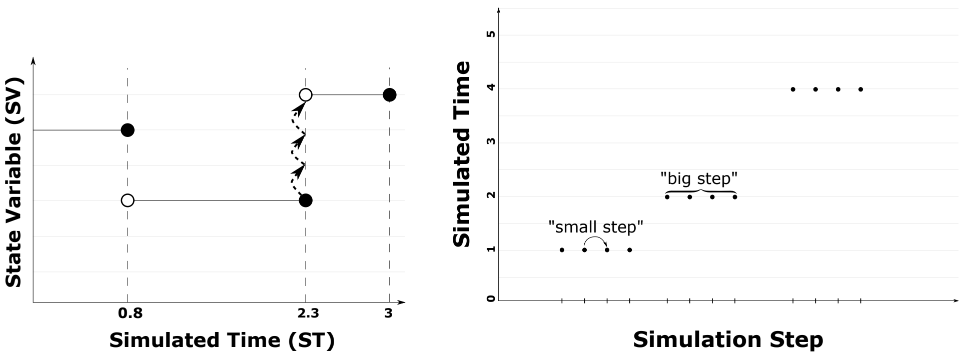
The previous section presented an overview of semantic variation points in simulation languages. We already mentioned for each variation point how they might affect debugging. This section divides useful debugging operations for simulations in different categories. Some are transpositions from code debugging operations, while others are simulation-specific.

The intent is to give an exhaustive overview of (potentially) useful debugging operations. The categories discussed in this section are presented in Figure 4.5. At a high level, we distinguish between *observation* of the state of simulation and *control* operations that influence the behaviour of the simulation process. Observation is implemented in the debugging-enhanced simulator and visualized by the debugging environment in the form of a model. Control operations are similarly implemented in the debugging-enhanced simulator and offered to the user in the debugging environment, in the form of interactive elements such as buttons. Within each category, we distinguish between operations relating to the steps of the simulation algorithm (discussed in Section 4.2.1), operations relating to the state of simulation (discussed in Section 4.2.2), operations relating to the simulated time (discussed in Section 4.2.3), and breakpoints (discussed in Section 4.2.4).

4.2.1 Stepping

In code debugging, stepping through the code is often used by the user to understand how the state of the system evolves during execution: it gives a detailed view of the program's behaviour. To transpose such operations to simulation debugging, we consider steps that give the user a view into the execution semantics of a simulator.

A formalism's simulator updates the state at certain points in simulated time (depending on the formalism's time semantics). Figure 4.6a depicts the evolution of the state (variable) over simulated time. Three *big steps* (state updates) are shown. A big step corresponds to an iteration of the simulation algorithm, and after it has completed successfully, the system is in a valid state. In between, however, a number of smaller computation steps may be involved. This is visualized in the figure by the dashed arrows at time 2.3 in the graph: each represents a phase in the computation of the next state. Another view is presented in Figure 4.6b, where one big step is broken up into a number of *small steps*. Simulated time stays constant in between small steps, and only increases after a big step has completed.



(a) Small steps for a discrete-event formalism.

(b) Another view of simulation steps: multiple steps occur at the same simulated time instant.

Figure 4.6: Small steps: a view into the simulation algorithm.

The results of state updates are available to the user after simulation if a state trace of the simulation is generated by the simulator. The phases of a big step (intermediate state changes), however, are not communicated to the user in a simulation environment, since they are implementation details of the simulator. Moreover, no control is offered to the user: during simulation, the simulator executes autonomously. Stepping operations needs to be implemented in a debugger, since stepping through a simulation's execution can provide valuable information to discover a defect.

We distinguish between two categories of steps. A big step and small step, as discussed above, are steps of the simulator: they are phases in the computation of the dynamics of the system. Such steps do not correspond to syntactic elements in the model, but rather, they correspond to an execution that involves the complete model state. And, while we have up to now considered only two such levels, a simulation algorithm can consist of many more: a small step can be broken up into multiple smaller steps, that themselves are broken up into even smaller steps. A different category of steps considers the syntactic scope of elements and attaches stepping semantics to certain syntactic constructs. This is the case, for example, in procedural languages, where a *step into* switches the context to the function being called. Conversely, a *step return* steps out of the current function. Such steps group a number of big steps and are only valid in certain contexts.

4.2.2 State

As explained, the system state evolves over (simulated) time during simulation. Usually, the modeller knows the initial state (since it is captured in the design model and potentially in a user-provided configuration that is used to initialize the runtime state of the simulation), as well as the end state (communicated after the simulation has finished). Inspecting the state during simulation is an important part of debugging, as it allows a user to see how the system evolves over (simulated) time. This requires a debugger to communicate intermediate runtime states to the user whenever a big step ends. And possibly, if more detailed information is required, intermediate changes performed by small steps can be communicated as well. The state of the system, however, might be (macroscopically)

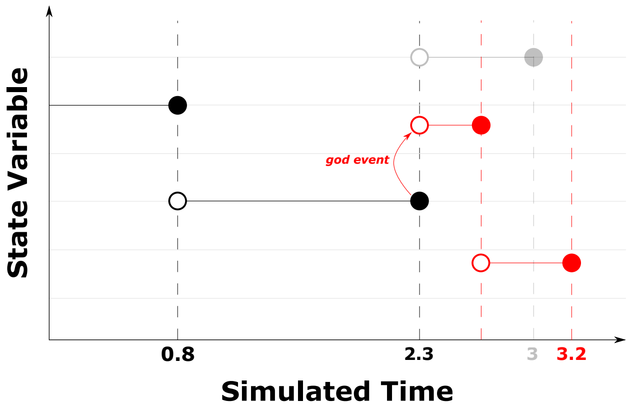


Figure 4.7: A god event changes the simulation state and influences future state updates.

inconsistent in between big steps. This is the reason it is not communicated during normal simulation: the inconsistent macroscopic states are not valid instances of the runtime language and thus cannot be displayed. Instead, small step information has to be visualized in a specialized language that extends the runtime language: a debugging language. This debugging language adds concepts relating to the intermediate simulation computations.

Aside from communicating the state at strategic points during simulation, a simulator can allow users to manually change the state of the system. This helps in refuting hypotheses related to the source of an error: changing the value of a suspect state variable and observing how the system dynamics change can, for example, rule out that particular value as being the cause of the error. We call such state changes *god events*, since they are an “outside force” not present in the original model that changes the state of the system during simulation. This is illustrated in Figure 4.7, where at simulated time instant 2.3 a god event manually alters the state. The state that originally resulted from the state update at that time instant is greyed out, and the new state is shown in red. The dynamics after the state change are different as well, as the simulator considers the god event as a regular state update and executes the semantics afterwards as usual.

4.2.3 Time

The notion of time plays a prominent role in model simulation. Simulated time differs from the wall-clock time: it is, as already explained, the internal clock of the simulator. Simulated time can, however, have a relation to the wall-clock time.

Program code is always executed as fast as possible (*i.e.*, the speed of the program is limited by the resources of the machine executing it and the operating system’s limitations). Simulations, however, can either be run as-fast-as-possible, or in (scaled) real-time. The latter is useful for simulating models of real-time systems which might be deployed as such on a real-time device. In this case, there is a linear relation between the wall-clock time and the simulated time. The different relations the simulated time can have to the wall-clock time are depicted in Figure 4.8 (adapted from [142]).

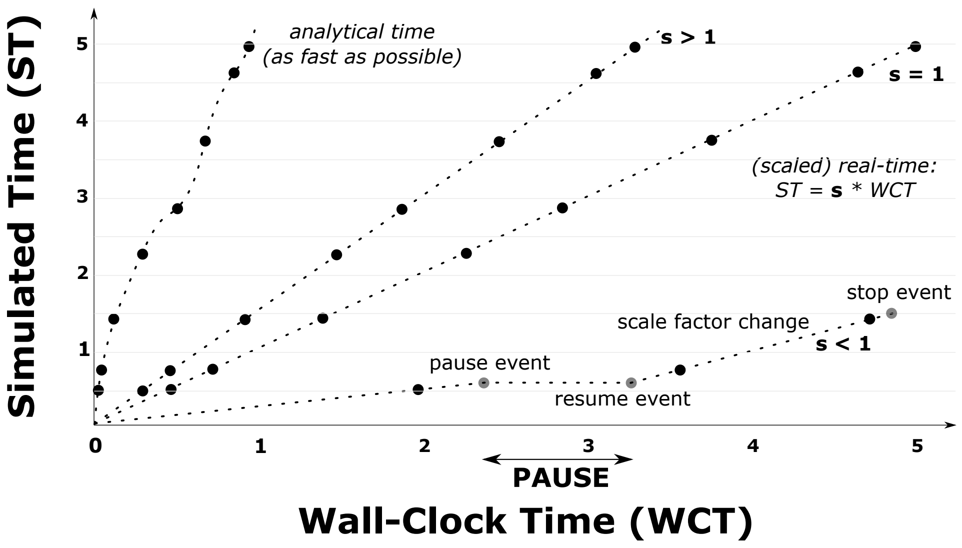


Figure 4.8: The different relations of simulated time to the wall-clock time.

In *as-fast-as-possible* simulation, there is no relation between simulated time and wall-clock time, meaning that simulated time is simply a variable in the simulator. In *real-time* simulation, simulated time is synchronized with the wall-clock time. This implies that the simulation steps have a hard real-time deadline (*i.e.*, the values of the runtime variables have to be computed before the wall-clock time reaches the value of the simulated time). A *scale factor* (in the figure depicted by the variable *s*) can be applied to speed up or slow down simulation, while maintaining the linear relation between simulated time and wall-clock time.

The mode of simulation has a direct effect on the debugging operations: both their availability and functioning.

- *State visualization* is only useful in real-time simulation. The intent of as-fast-as-possible simulation is to execute the simulation quickly, and having to visualize the state in between would put too high of a burden on the complete system's execution speed. Also, since the simulator executes as fast as the underlying platform allows, the updates to the runtime model can occur very fast. Understanding the execution by trying to interpret an uncontrollably fast visualization of the state updates is difficult.

- *Pausing* a simulation in real-time simulation has different behaviour than pausing a simulation in as-fast-as-possible simulation. Figure 4.9 depicts the difference between the two modes. In as-fast-as-possible mode (visualized by the stepwise function), the computations of the next state (represented by the horizontal parts in the function) are executed one after the other, without any waiting in between. This means that if a pause is requested (denoted by the red vertical bar with "pause" next to it), the simulation will be paused only after completion of that big step computation. Halting immediately might otherwise leave the system in a (macroscopically) inconsistent state. In real-time mode, the time needed to compute the next state is still there, but now the simulator will wait in between these computation periods to synchronize

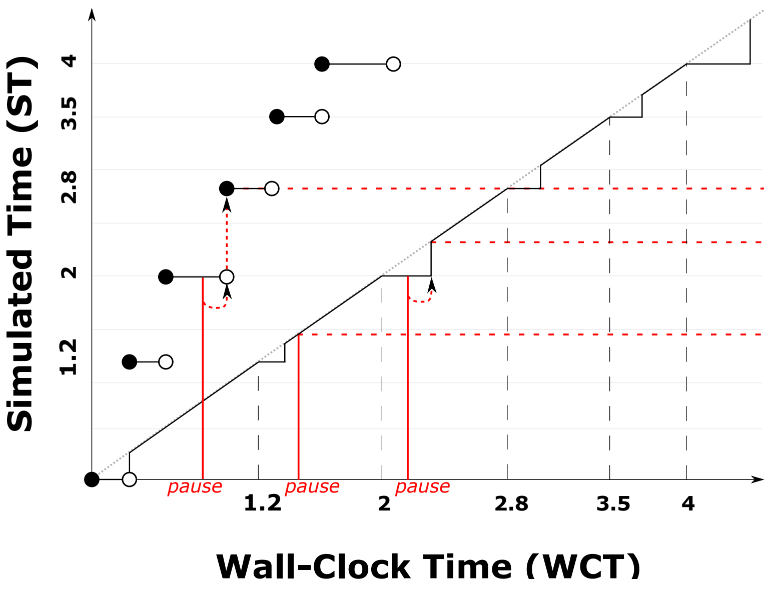


Figure 4.9: Pausing: difference between as-fast-as-possible and real-time simulation.

the simulated time with the wall-clock time. This is represented by the continuous function, which tries to follow the ideally synchronized line, represented by the grey dotted line. When a computation is performed, the wall-clock time advances, thus desynchronizing the simulated time from the wall-clock time. This is represented by the horizontal parts in the function. When the computation is finished, the simulator will synchronize both times immediately, as depicted by the vertical parts. If a pause is requested during a waiting period, the simulator immediately pauses. The result is that the system will be in the "current" state, and not the "next", as was the case for as-fast-as-possible mode. The simulated time will be in between the previous transition time and the next. Due to this difference, an as-fast-as-possible simulation cannot pause at times in between two different simulated times. On the other hand, real-time simulation can pause at virtually every point in simulated time. The notable exception being the time at which transition functions are being computed.

Being able to control the relation between simulated time and the wall-clock time is an important feature of a debugger. It allows users to inspect the execution semantics of real-time systems (and, if necessary, speeding it up or slowing it down while retaining the linear relation) visually, but also allows to quickly jump to a point of interest (using pausing or breakpoints, discussed in the next subsection).

4.2.4 Breakpoints

A simulation can be paused manually, which halts the simulation in a (macroscopically) consistent state. As discussed above, the behaviour of the pause operation can differ depending on the execution mode of the simulator.

Pausing the simulation manually at a point of interest might be difficult, if the user is only able to decide when to pause based on the state updates received from the simulator. Alternatively, breakpoints allow the user to automatically pause the execution when a condition is satisfied. Breakpoints are transposed operations from code debugging, where breakpoints are used to pause the program when a specific line of code is reached. Additionally, a breakpoint can be augmented with a condition on the runtime state. Model debuggers can expose, similarly, a way of setting breakpoints that depend on the simulation state and the simulated time value.

In this thesis, we define a breakpoint as a function $Br(state, time)$ which returns *True* if the simulation should pause, and *False* otherwise. This includes the traditional notion of a programming code breakpoint, since the currently executing line (the *program counter*) is part of the runtime state of the program. A breakpoint has a name, to identify it when it was triggered. It can also be enabled or disabled; additionally, it can be automatically disabled when it triggers. This is useful if the breakpoint's condition will remain true after it has triggered once. In that case, if it would not be disabled automatically, it would continuously trigger afterwards.

Breakpoints are evaluated when the system is in a consistent state. For simulation systems, we have established that the state is consistent after a "step" (simulation iteration) has ended (see Figure 5.14). A "big step" can consist of multiple smaller steps, but after such a smaller step, the system might be in a macroscopically inconsistent state. Therefore, breakpoints (and pauses) can only occur after a big step has ended. This is different from programming code breakpoints, which are checked only when a certain line of code is reached. This optimization is enabled by the operating system, which allows the program to be instrumented with breakpoint instructions that interrupt the normal flow of control. In general, a simulator cannot rely on such operating system functions, since the complete simulation state has to be checked. There is, however, a possibility for "syntactic sugar" to make it easier to place breakpoints at interesting locations. To do this, syntactic constructs of the runtime model can be reused. For example, if a language has an explicit notion of "state", a breakpoint can be attached to that syntactic construct to only break when that state was entered. Or, if the language has the capability to evaluate expressions on state variables natively, a breakpoint can be attached to such expressions to break only when the expression becomes true. But, the breakpoint still only can access the full simulation state and simulation time in order to decide whether or not to pause. Many domain-specific notations are possible for breakpoints; in this thesis, some possibilities are explored (see Chapter 5 for examples), but no attempt is made to be exhaustive.

A more elaborate possibility for breakpoint conditions is to allow conditions on the full simulation trace. This enables the specification of temporal properties, that, when they evaluate to true, halt the simulation. These temporal properties can be compared to the properties that can be specified using the ProMoBox approach. In this case, however, a user specifies properties that should *not* hold, instead of those that should. An interesting use case is the interplay of property checking, followed by debugging, where a property that was not satisfied by the system is negated and set as a breakpoint condition. The simulation can then run until the point where the property is not satisfied, at which point the interactive debugging session is started. We leave such breakpoint conditions for future work, and focus on breakpoints defined on the state and time of the simulation. Breakpoints are always evaluated after a big step has ended (and the system is in a consistent state).

4.3 De- and Reconstruction of Model Simulators

Developing debuggers for modelling languages, taking into account their diverse semantics, is an inherently complex task. The interplay of formalism execution semantics, different notions of simulated time such as (scaled) real-time and as-fast-as-possible execution, the semantics of debugging operations, as well as user interaction through an interface are all challenging to capture and implement correctly. Specifically, traditional software engineering methods where such debuggers would be implemented using a programming language, fall short. The essential complexity of the debugger is:

- its timed behaviour that implements the real-time execution semantics of the formalism;
- its interaction with the user in two ways (a user can execute debugging operations, interrupting the normal flow of the simulator, and the debugger provides updates to the user through the debugging interface);
- its autonomous behaviour, implementing the semantics of the modelling language.

The combination of these behavioural properties is not easily expressed using programming languages, as it is inherently concurrent and the interleaving of the different threads of control is difficult to manage. Instead, we apply the multi-paradigm philosophy and model the behaviour of the debugger at the most appropriate level of abstraction, using the most appropriate formalism.

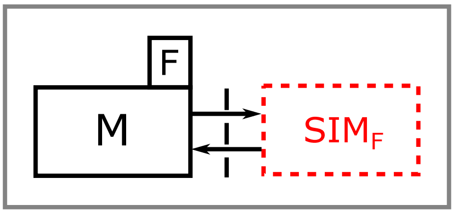
ALGORITHM 1: A generic simulation algorithm.

```
Input: Model to simulate ($M$), parameters ($params$).
1 $time, state \leftarrow initialize(M, params)$;
2 while not endCondition(M, time, state) do
3     $state \leftarrow simulatorStep(M, state)$;
4     $time \leftarrow incTime(M, time)$;
5 end
6 return $state, time$;
```

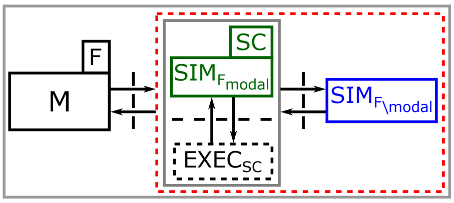
Before we model the behaviour of debuggers, we need to examine their structure. At a high level of abstraction, simulation algorithms are very similar, as they go through a set of phases:

1. *Initialization* of simulation time and the simulation state;
2. *Execution* of simulation 'steps' until an end condition is satisfied (the core of the algorithm, where a new state is computed based on the previous one, and the simulation time is advanced);
3. *Finalization* where, for example, the final state of the simulation and the time at which it ended is communicated to the user.

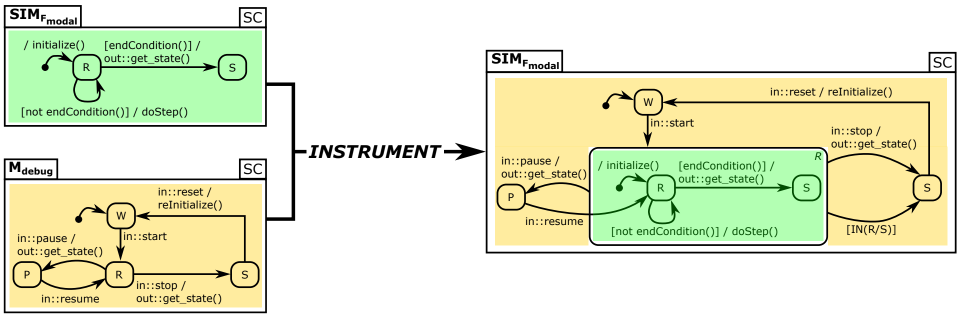
Algorithm 1 presents a generic simulation algorithm in pseudocode which encodes these semantics. Most often, simulators are implemented in program code. And, while they aren't necessarily decomposed in the same functions as Algorithm 1, they can be converted



(a) A model is simulated by an appropriate simulator for its formalism.



(b) The simulator is deconstructed by extracting its modal part.


SIM_Fmodal
SC
/ initialize()
[endCondition()] / out::get_state()
R
S
[not endCondition()] / doStep()
M_debug
SC
in::reset / reInitialize()
W
in::start
in::pause / out::get_state()
P
R
S
in::resume
in::stop / out::get_state()
INSTRUMENT
SIM_Fmodal
SC
W
in::reset / reInitialize()
in::start
in::stop / out::get_state()
in::pause / out::get_state()
/ initialize()
[endCondition()] / out::get_state()
R
P
R
S
S
in::resume
[not endCondition()] / doStep()
[IN(R/S)]

(c) The modal part of the simulator is instrumented with debugging operations

Figure 4.10: De- and reconstruction of simulators.

into a similar form. This is the first step towards enhancing the algorithm with debugging support. The simulation algorithm is decomposed into the following functions:

- *initialize* initializes the state of the system. This returns the initial state s_0, conforming to the runtime language, and t_0, the simulated time at which simulation starts.
- *endCondition* returns true when the simulation is finished. It is a user-specified function that receives the current state of the simulation (including the current simulated time) as a parameter. Complex conditions can be modelled that combine multiple state variables, but we do not consider temporal conditions on a state trace. Such temporal conditions can only be modelled if the function has a memory component, or if a full state trace is saved and passed to the end condition function.
- *simulatorStep* executes one step of the simulation algorithm, changing the state of the system.
- *incTime* increases the simulated time according to the time semantics of the formalism.

Any simulation algorithm for the types of formalisms discussed previously in this chapter (Figure 4.10a) can be written in the form presented in Algorithm 1. This form separates the *modal* part of the simulator (the flow of control, mainly consisting of the "main simulation loop") from the *non-modal* part.

This modal part can be "lifted out" and modelled in an appropriate formalism, for which we choose Statecharts. Figure 4.11 shows the modal part of the generic simulator.

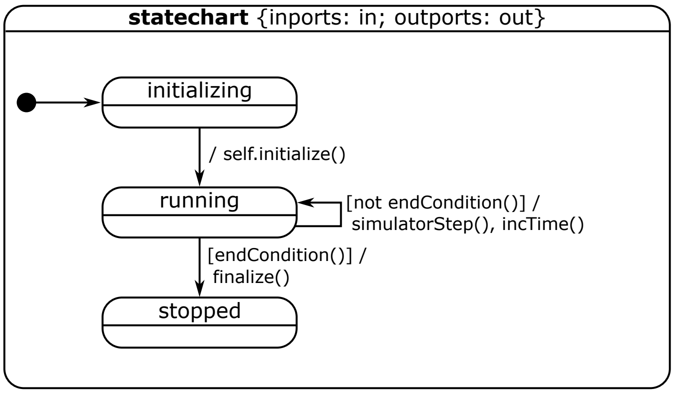

statechart {inports: in; outports: out}
initializing
/ self.initialize()
running
[not endCondition()] / simulatorStep(), incTime()
[endCondition()] / finalize()
stopped

Figure 4.11: The modal part of the generic algorithm.

When this model is combined with an appropriate executor that implements the semantics of the Statecharts formalism, and combined with the non-modal part of the simulator (implemented in the functions called in the actions of the Statecharts transitions), we obtain a version of the simulator that implements identical semantics. It is now simply broken up in two parts (Figure 4.10b). The choice of which behaviour is model and which behaviour is non-modal is up to the language engineer tasked with splitting up the behaviour. But, there are a number of criteria that guide the decisions:

- Any behaviour that is to be *interrupted* in the debugging-enhanced simulator should be modelled in the modal part of the simulator. For example, after every big step, a pause can occur. Or, if the user is to given control beyond that (in the form of small steps), the control flow of the algorithm is broken up into smaller components that can be interrupted as well.

- Any behaviour which has *time delays* (for real-time simulation) should be modelled in the modal part of the simulator.

- Any non-interruptible *computations* belong in the non-modal part of the simulator.

These criteria aid the language engineer in dividing the algorithm in its modal and non-modal components.

In Figure 4.10c, the next step in creating a simulator which supports debugging is shown. We *merge* the modal part of the simulator behaviour model with a model capturing the debugging operations we want to add. This results in an instrumented model of the modal behaviour of the simulator.

The last step consists of replacing the original modal part of the simulator with its instrumented version. For continuity reasons, the behaviour of the simulator remains unchanged if the user does not make use of the debugging functionality. Extra behaviour has been added, but running the simulator as before is still possible. In the example shown, the debugger includes the concepts of *start*, *pause*, *resume*, and *stop*. The simulator only has two states: *R* (*Running*), and *S* (*Stopped*). This is a trivial (and fictional) example, but it demonstrates the process which we call de- and reconstruction of the simulator.

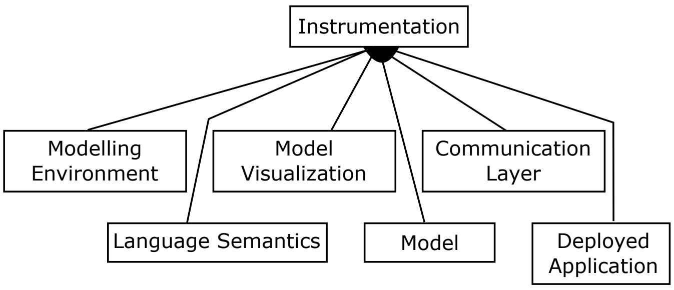


Figure 4.12: The artefacts that can be instrumented with debugging support.

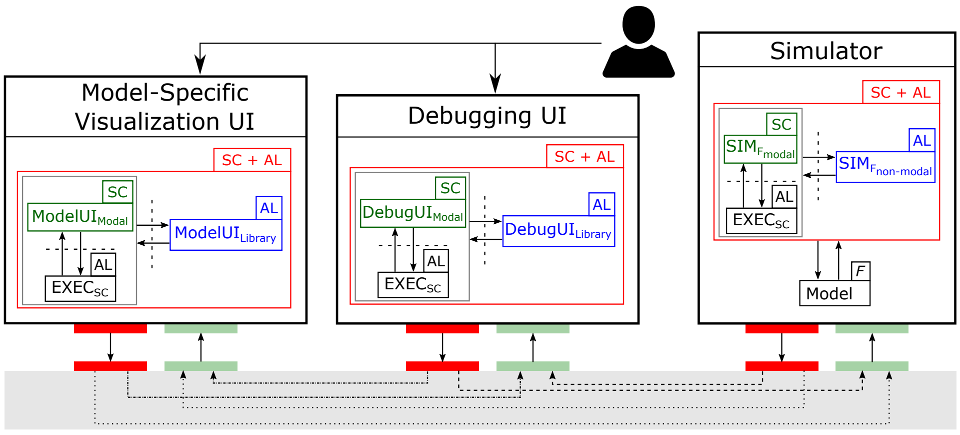


Figure 4.13: The generic architecture of a debugging environment for formalism *F*.

The result of reconstructing the simulator is an instrumented version of the original simulator, enriched with debugging capabilities. From this model, a debugging-enhanced simulator for formalism *F* can be automatically generated using a Statecharts compiler. The debugging-enhanced simulator has an extended interface, which allows a user to control and observe the simulation process using debugging operations. In the next section, the debugger is placed in an architecture to implement a debugging environment for the formalism.

4.4 Architecture

In Figure 4.12, we list the components that were constructed during the modelling and simulation workflow (see Figure 2.9) as candidates to be instrumented with debugging support. We consider all components, except the deployed application, as the scope of our work is on model simulation, not deployment. To debug deployed applications would require to instrument their code with appropriate code that talks back to a debugging environment during execution, which has been covered in numerous related works (see Chapter 3). Although work remains to be done on integrating these approaches with our proposed solution, we focus on model simulation. These components are connected in an architecture of our solution, and shown in Figure 4.13. It consists of three components, whose behaviour is specified in Statecharts models (and definitions of their input and output sets):

- The simulator, whose behaviour is described by

  $$S_{SIM_F} = < SIM_{F_{Modal}}, SIM_{F_{Mem}}, SIM_{F_{Library}}, X_{SIM_F}, Y_{SIM_F} >$$

- The debugging user interface, whose behaviour is described by

  $$S_{DebugUI} = < DebugUI_{Modal}, DebugUI_{Mem}, DebugUI_{Library}, X_{DebugUI}, Y_{DebugUI} >$$

- The model-specific visualization interface, whose behaviour is described by

  $$S_{ModelUI} = < ModelUI_{Modal}, ModelUI_{Mem}, ModelUI_{Library}, X_{ModelUI}, Y_{ModelUI} >$$

  which is instrumented with debugging operations.

The first two models, S_{SIM_F} and $S_{DebugUI}$, have to be defined once and can be reused to debug any model created in the simulator's formalism. The model S_{SIM_F} is obtained from an existing simulator by de- and reconstructing it (see the previous section). The model $S_{DebugUI}$ is a behavioural description of a debugging environment, which is generated from the definition of a debugging language, similar to the design, runtime, and output languages. We rely on metamodelling environments such as AToMPM [192] to provide the necessary infrastructure to obtain such a model from the definition of the language. In the architecture presented in Figure 4.13, we only consider the generated behavioural description in the form of a Statecharts model, and not the language definition. This language definition is obtained by extending the runtime sub-language of the modelling language, and will be discussed in the next chapter. The third model, $S_{ModelUI}$, is model-specific, and is created by a user to visualize the state of a particular simulation system. This is particularly useful in dynamic-structure simulations, for example those that model agent-like behaviour. The model-specific visualization interface can visualize a runtime state of the simulation (a model conforming to the runtime sub-language) and can be repurposed for debugging tasks, if it is instrumented appropriately. We assume its behaviour is specified using Statecharts, which allows us to reuse the techniques presented in the previous section to instrument the model with debugging operations.

A *communication layer* is responsible for the communication between components. We do not specify here what implementation technology should be used, but any (asynchronous) inter-process communication mechanism, such as sockets, are suitable. Note that our architecture does not keep the simulator, model-specific visualization, and the debugging interface synchronized. Synchronization is necessary in the case of (scaled) real-time simulation, since only in this mode the user can observe state changes while the simulation is running. We observe that the simulator is the most computation-intensive component in our architecture, while the visual interfaces (almost trivially) perform an update when a state change is received. Synchronization is therefore unnecessary: if the simulator is unable to compute the next state of the system in time during scaled real-time simulation, the scale factor the user chose is too small. This can be detected by the simulator and a notification can be given to the user. The communication layer is furthermore responsible for translating events sent by components to events that are accepted by another component. There are six translation functions:

- $Z_{ModelUI,SIM} : Y_{ModelUI} \rightarrow X_{SIM}$ translates output events of the model-specific visualization UI (when the user interacts with the simulation by pressing a key, for example) to input events of the simulator. These correspond to (real-time) interrupts that are propagated to the model.

- $Z_{SIM,ModelUI} : Y_{SIM} \rightarrow X_{ModelUI}$ translates output events of the simulator to input events of the model-specific visualization. These correspond to output events generated by the model as it is being simulated, which trigger a change in the visualization.

- $Z_{DebugUI,SIM} : Y_{DebugUI} \rightarrow X_{SIM}$ translates output events of the debugging UI (*e.g.*, when the user presses a button in the debugging UI) to input events of the simulator (representing debugging operations, such as a request to pause the simulation).

- $Z_{SIM,DebugUI} : Y_{SIM} \rightarrow X_{DebugUI}$ translates output events of the simulator to input events of the debugging UI. These correspond to debugging messages and information that needs to be visualized in the debugging UI, such as state changes.

- $Z_{DebugUI,ModelUI} : Y_{DebugUI} \rightarrow X_{ModelUI}$ translates output events of the debugging UI to input events of the model-specific visualization UI.

- $Z_{ModelUI,DebugUI} : Y_{ModelUI} \rightarrow X_{DebugUI}$ translates output events of the model-specific visualization UI to input events of the debugging UI.

By tying together the debugging-enhanced simulator, debugging environment, and model-specific visualization, we can build advanced debugging environments for any formalism. The next section explains the workflow for building such environments.

4.5 Workflow

In this section, we present a workflow for constructing the artefacts in the architecture presented in the previous section. We start from an existing simulation kernel, a modelling and simulation environment without debugging support, and a model-specific visualization UI without debugging support, and enhance them with debugging support.

The workflow is modelled using an FTG+PM language in Figure 4.14. There are four tasks: de- and reconstructing the simulation kernel (explained in Section 4.3), instrumenting the modelling environment, instrumenting the model-specific visualization UI, and creating a communication layer.

Once the debugging enhanced simulation kernel is developed, an interface for interacting with it is needed. The debugging interface allows the user to control the simulation algorithm using the set of debugging operations defined in Section 4.2 and implemented in Section 4.3. This debugging interface is an extension of an already existing modelling (and simulation) interface. We assume the design sub-language, runtime sub-language, and output sub-language of the modelling language for which debugging is implemented already exist—this means a model in the language can be designed, and its runtime state can be displayed, as well as its simulation result. We reuse these parts of the modelling

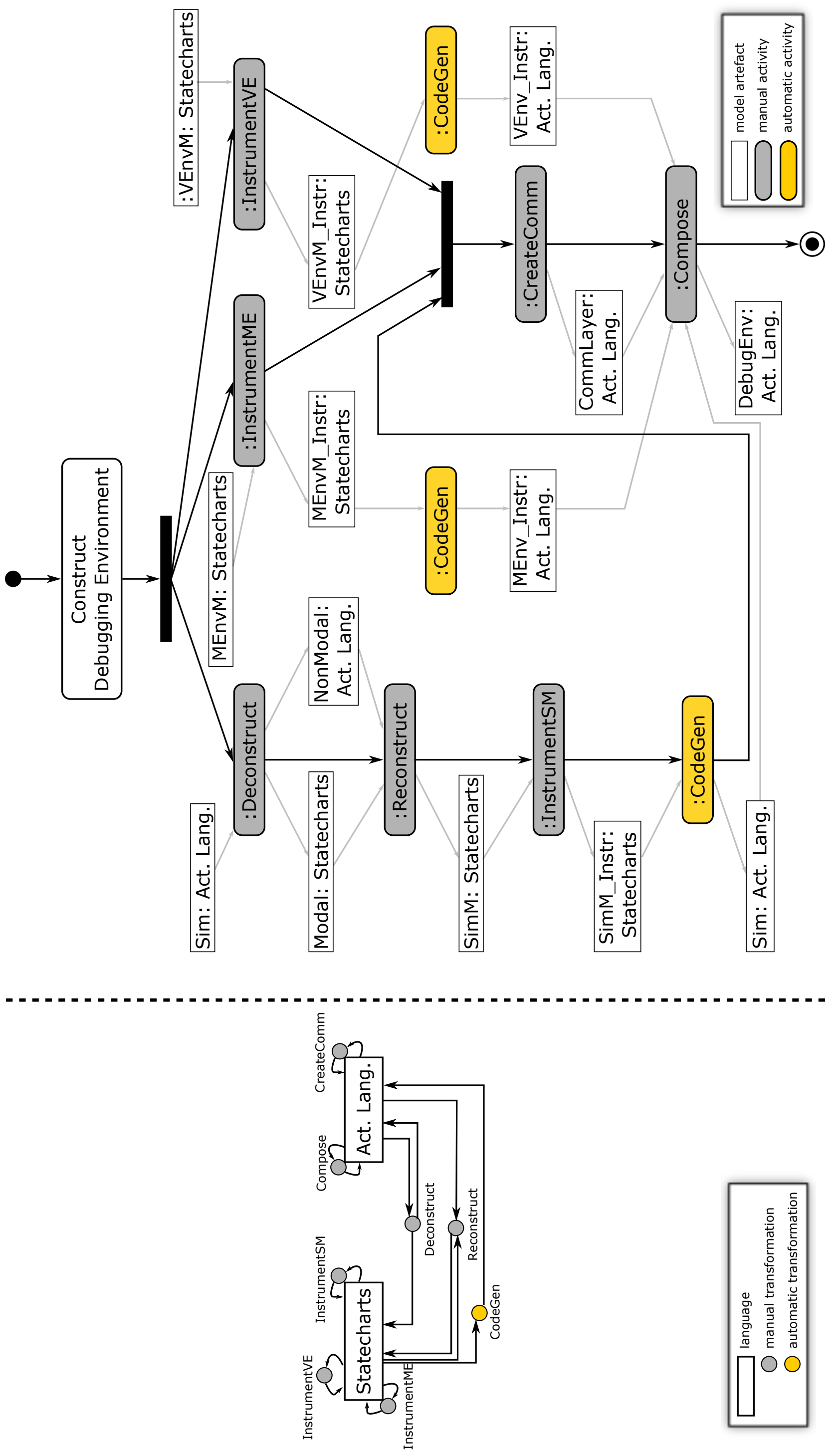


Figure 4.14: The workflow for constructing a debugging environment.

environment to implement debugging. We need to add a specialized debugging language, however. Since we instrument the simulator of the language with debugging support, the control and observation operations have to be available in the debugging environment. The state of the simulation (computed after every *big step*) can already be displayed. For each lower-level step (*small step*), however, a debugging-specific visualization has to be added in the form of an element in a debugging language. This new language consists of a number of elements that visualize the intermediate states of the simulation (which are not necessarily well-defined runtime states of the formalism, and therefore, not present in the runtime language). Additionally, *breakpoints* require a visual representation as well. The conditions on which breakpoints pause the simulation naturally differ depending on the language, but often breakpoints can be scoped to only consider parts of the model. This can be specified in a visual debugging environment by connecting breakpoints to those elements.

To offer control to users, they need a way of sending debugging commands from the user interface. In graphical modelling environments, actions are often presented to the user as buttons in a toolbar. A debugging toolbar is added to graphical modelling environments to allow users to communicate with the instrumented simulator. Since this is a 1-1 mapping of the debugger's interface to buttons in a toolbar, they can be trivially added to the interface.

A model-specific visualization can also be reused for debugging purposes. While de- and reconstructing the simulator and implementing a debugging interface are tasks a simulation expert has to perform once, after which they can be reused, model-specific visualizations need to be adapted by the domain expert that created them. For this to work, the domain expert has to instrument the Statecharts model of the model-specific visualization UI with appropriate transitions that raise events on its output port. A translation function, which translates these events to input events of the debugging UI also has to be specified. The same tasks have to be performed if the visualization UI has to respond to events from the instrumented simulation kernel, such as state updates.

Summary

This section explains how model debugging environments can be constructed by extending the state-of-the-art in language engineering. As a starting point, we consider existing techniques for constructing design, simulation, and replay environments. From there, we present an instrumentation approach for adding debugging support to an existing implementation of the semantics of a modelling language in the form of a simulator. This debugging-enhanced simulator is placed in an architecture that connects it with a debugging environment (generated from the definition of a debugging language, which is an extension of the modelling language's runtime language) and other visualizations that can assist in the debugging process. The debugging environment construction is guided by a workflow. In the next chapter, we use the workflow, architecture, and instrumentation techniques to develop debugging environments for a number of representative formalisms, demonstrating that our approach is applicable for a wide variety of languages.

Chapter 5

Representative Formalisms

This chapter applies the techniques presented in the previous chapter to a number of carefully chosen formalisms, to demonstrate feasibility. Each language is classified according to its semantic properties that were discussed in Section 4.1. We cover a large variety of semantically different languages. For each language, we explain their syntax (used to model a design in the system) and their semantics (implemented operationally in a simulator or translationally by a transformation that maps onto a formalism with known semantics). We present each language's simulator in pseudocode, and construct a set of useful debugging operations. To implement the debugging operators, we de- and reconstruct the simulator and instrument it. We then connect the simulator to a (graphical) debugging environment that allows a user to interact with the debugger and display intermediate states of the simulation.

Demonstrations, artefacts, and reports for all developed tools are available at `http://msdl.cs.mcgill.ca/people/simonvm/phd_thesis`.

Structure We present eight formalisms in this chapter. The goal of these eight formalisms is to cover as much as possible of the semantic variation points identified in Section 4.1.

- Section 5.1 implements debugging for an explicitly modelled action language formalism. This formalism:
  - is **procedural** (functions can call other functions);
  - is **untimed**;
  - is **deterministic**;
  - has **static structure**;
  - allows for **environment interaction** (through input/output functions).
- Section 5.2 implements debugging for Causal Block Diagrams, a dataflow language in which mathematical equations can be modelled. This formalism:

  - allows to model **algebraic loops** (that need a specific schedule to be computed in a consistent manner);
  - implements **discrete-time** or **continuous-time** semantics (depending on the types of blocks used in the instance models);
  - is **deterministic**;
  - has **static structure**.

- Section 5.3 implements debugging for Parallel DEVS, a discrete-event formalism that is decomposed hierarchically into components that communicate using events. This formalism:
  - implements **discrete-event** semantics;
  - is **deterministic**;
  - has **static structure**.
- Section 5.4 implements debugging for Statecharts, a discrete-event formalism where *states* are the main abstraction, which are composed hierarchically and orthogonally. This formalism:
  - implements **discrete-event** semantics;
  - is **deterministic**;
  - has **static structure**;
  - allows for **environment interaction** (through input/output events).
- Section 5.5 implements debugging for Petrinets. This formalism:
  - is **untimed**;
  - is **non-deterministic**;
  - has **static structure**.
- Section 5.6 implements debugging for Dynamic Structure DEVS, a dynamic-structure extension of Parallel DEVS. This formalism:
  - implements **discrete-event** semantics;
  - is **deterministic**;
  - has **dynamic structure**.
- Section 5.7 implements debugging for a hybrid formalism, composed of a discrete-event and a continuous-time dataflow formalism. This formalism:
  - is **hybrid** (through **embedding**).
  - allows to model **algebraic loops** (that need a specific schedule to be computed in a consistent manner);

  - implements **discrete-event** semantics, combined with **discrete-** or **continuous-time** semantics (depending on the types of blocks used in the instance models);
  - is **deterministic**;
  - has **static structure**.

- Section 5.8 implements debugging for a domain-specific language whose semantics are implemented translationally, by mapping onto a formalism with known semantics. This formalism:
  - implements **discrete-event** semantics;
  - is **deterministic**;
  - has **static structure**.

  Because of the translational definition of its semantics, the debug operations performed on the domain-specific level have to be translated to the target formalism, and the traces produced by the target formalism have to be translated to the domain-specific level.

5.1 Action Language

Action language is a formalism similar to programming languages. It is often used to express algorithmic, imperative code, when it is most appropriate. In this section, a debugger for a procedural, sequential action language is constructed. We choose the action language of the Modelverse [201] for the following reasons:

- It is minimal, but contains all essential functionality of a procedural, sequential language. We avoid accidental complexity and can focus on the essential complexity of constructing a debugger for such a language.
- Its syntax and semantics are explicitly modelled. The syntax is, similar to other modelling languages, expressed in a metamodel. Its semantics are expressed using a set of graph transformation rules that modify the structure of the execution state.
- Its definition is readily available, as well as its executor. This facilitates the instrumentation of the executor with debugging support.

The action language is a causal, untimed, deterministic, static-structure, procedural language.

In the following subsections, we explain the syntax and semantics of the action language and the execution support in the Modelverse. We then present a set of useful debugging operations, inspired by code debugging operations. These debugging operations are implemented using the de- and reconstruction technique and a (textual) debugging environment is generated.

5.1.1 Syntax and Semantics

The action language of the Modelverse is a minimal procedural, sequentially executed language. A program in the Modelverse can consist of:

- *If*-statements that execute one (or none) of its child code blocks based on the evaluation of its conditions.
- *While*-loops that execute a code block as long as its condition is satisfied.
- *Break* statements that break out of their enclosing loop by jumping to the instruction that follows the loop.
- *Continue* statements that jump to the condition evaluation phase of its enclosing loop.
- *Return* statements that return from a function. A value can be returned if the function is declared to have a return type.
- *Function definitions* that encapsulate a block of action code. They have a number of typed parameters, and a return type.
- *Function calls* that refer to a previously defined function by name, and provide values for the called function's parameters.
- *Variable declarations*, either local or global. A variable is declared by stating its name and type. After declaration, it has to be defined by assigning a value.
- *Assignment* statements that assign the result of evaluating an expression to a (previously declared) variable.
- *Constants*, of type integer, float, boolean, string, or action code.
- Blocking user input/output functions.

```
include "io.alh"

Integer function fib(param : Integer):
  if (param <= 2):
    return 1!
  else:
    return (fib(param - 1) + fib(param - 2))!

Void function main():
  while(True):
    output(fib(input()))
```

Listing 5.1: An example action language model modelling a function to compute Fibonacci numbers.

An example action language model in a textual syntax is shown in Listing 5.1. The model defines a recursive function to compute Fibonacci numbers. Its main function listens for user input (a number) and outputs the result (the Fibonacci number corresponding to the input parameter) back to the user. An action language model can define multiple functions, as is the case here. When it is executed, the *main* function is called if it exists. If it does not, then the first defined function is called.

ALGORITHM 2: The main loop of the Modelverse.

```
while $True$ do
    for $t \in getTasks()$ do
        while $isActive(t)$ and $iters < getMaxIters()$ do
            $executeRule(t)$;
            $iters \leftarrow iters + 1$
        end
    end
end
```

The Modelverse [207] is a model repository and model management kernel. It is based on a number of axioms, one of them being that every element is explicitly modelled. The execution of the Modelverse is driven by a number of graph transformation rules that are continuously executing user-defined programs (called *tasks*), which are stored as graph structures. A compiler is responsible for translating a textual model definition to the graph structure required by the Modelverse. The main execution loop of the Modelverse is shown in Algorithm 2. It is an infinite loop which continuously advances the state of all its tasks that have an instruction that needs to be executed. For each active task, a number of rules associated to its currently executing statement are executed. This number is constrained by a constant and allows the Modelverse to more efficiently execute tasks: if only one rule were executed each time, the number of task switches would be high and consequently, the performance low. Executing a rule in the Modelverse corresponds to executing a "phase" of an instruction. An instruction (such as the evaluation of an expression) is broken up into these smaller, more primitive phases, as they need to be modelled as graph rewriting rules. For example, assigning an expression to a variable first needs to evaluate the expression, then it needs to retrieve the variable's location, and finally, assign the new value to it.

The rules that can be executed are listed below. For an in-depth discussion on the execution semantics of the Modelverse, see [202].

- Execute a *continue* statement (in a *while*-loop). Consists of one phase, which sets the instruction counter to the instruction that evaluates the condition of the *while*-loop.

- Execute a *break* statement (in a *while*-loop). Consists of one phase, which sets the instruction counter to the instruction following the *while*-loop.

- Execute an *if*-statement. Consists of two phases: first, it moves the instruction pointer to the condition, then, it evaluates the condition and correctly sets the instruction pointer to either the first instruction of the *true*-block, and otherwise to the first instruction of the *false*-block.

- Execute a *while*-statement. Consists of two phases: first, it moves the instruction pointer to the condition, then, it evaluates the condition and correctly sets the instruction pointer to either the first instruction of the *while*-body, and otherwise to the first instruction following the *while*-body.

- Resolve a variable. Consists of one phase: find the variable's value and move the instruction pointer to the next instruction.

- Execute an assignment. Consists of three phases: first, it moves the instruction pointer to the value expression, then, it resolves the variable, then, it evaluates the value expression, and last, it assigns the value to the variable and sets the instruction pointer to the next instruction.

- Execute a *return* statement. Consists of two phases: first, it moves the instruction pointer to the value expression, then, it evaluates the value expression, retrieves the previous frame from the stack to store the value, and last, it pops the current frame from the stack.

- Evaluate a constant. Consists of one phase that simply returns the value of the constant.

- Call a function. Consists of a number of phases depending on the number of parameters: first, it checks whether there are any parameters and if so, executes a phase for each parameter to evaluate its expression. Last, it initializes the function's frame and moves the instruction pointer to the first expression of the function.

The compiler built into the Modelverse already attaches to the nodes in the compiled abstract syntax graph its line information, storing at which line and character position in the source file the statement was defined. This information can be read and printed at times when it is needed, for example if an exception occurs. Of course, in terms of debugging support, this is limited, since we can only observe where the instruction pointer currently is. Moreover, while a call stack is internally maintained by the Modelverse, to which a stack frame is pushed each time a function is called, and popped from when the current function returns, that information is not used when printing out errors.

5.1.2 Debugging Operations

We take inspiration from a well-established, advanced and actively developed debugger: the GNU debugger (gdb)[1]. It allows developers to debug programs written in C or C++, and has partial support for a number of other languages. This section presents an overview of its capabilities, as it is a representative example of a code debugger. We do not list all of gbd's features, since we are interested only in the operations for interactive debugging. We focus on its operations for taking control over the program's control flow, how state can be inspected, how state can be changed, and its record/replay capabilities.

Taking Control

A program is placed under control of gdb by executing the command *gdb program*, where 'program' is the program to debug. Alternatively, gdb can be attached to an already running process. When gdb is started, it takes control over the process and a shell opens in which the user can type commands. From that point on, the user can control the execution of the program using multiple debugging operations, explained in the paragraphs below.

[1]https://www.gnu.org/software/gdb/

Breakpoints Breakpoints are automatic pauses on a state condition. They are set with the *break* command. If no parameters are passed, a breakpoint is placed at the next instruction to be executed in the current stack frame. A *location* (function name, line number, or an address of an instruction) can be passed as parameter. Optionally, a condition can be passed. In that case, the condition must evaluate to true in order for the breakpoint to trigger. A *tbreak* is a special kind of breakpoint which is removed once it triggers. It is possible to list all breakpoints, and to delete or disable them.

Watchpoints automatically pause the program when the value of some variable changes. Typically, a user monitors whether a particular variable's value changes by executing *watch foo*, where 'foo' is the symbolic name for the variable. The *rwatch* command breaks the program when a variable is read. The *awatch* command breaks the program when a variable is read or written. It is possible to list all watchpoints, and to delete or disable them.

Catchpoints automatically pause the program when certain types of events occur. This includes system calls, exceptions, and the loading of shared libraries.

Continuing If a program is paused, execution can be resumed in several ways.

- The *continue* command resumes continuous execution.
- The *step* command resumes execution until the program reaches a different source line. Optionally, the user can specify how many steps are to be executed.
- The *next* command resumes execution until the program reaches a different source line in the current stack frame. Optionally, the user can specify how many steps are to be executed.
- The *finish* command resumes execution until the currently executing function returns.
- The *until* command resumes execution until a source line past the current line in the current stack frame is reached. Optionally, the user can specify the location of the line at which to stop execution.
- The *advance* command resumes execution up to a given location (not necessarily in the same stack frame).
- The *stepi* command executes one machine instruction. Optionally, the user can specify how many steps are to be executed.
- The *nexti* command executes one machine instruction, but if it is a function call, it proceeds until the function returns. Optionally, the user can specify how many steps are to be executed.
- The *reverse-continue* starts reversing the program's execution.
- The *reverse-step* command reverses the program's execution until the program reaches a different source line. Optionally, the user can specify how many steps are to be executed.

- The *reverse-next* command reverses the program's execution to the beginning of the previous line executed in the current stack frame. Optionally, the user can specify how many steps are to be executed.
- The *reverse-finish* command reverses the program's execution until the point at which the currently executing function was called.
- The *reverse-stepi* command executes one machine instruction in reverse. Optionally, the user can specify how many steps are to be executed.
- The *reverse-nexti* command executes one machine instruction in reverse, but if it is a return from a function, it reverses the program's execution until the point at which the function was called. Optionally, the user can specify how many steps are to be executed.

Inspecting the State

The state of the system is made up of the variable's values (data) and the execution state (frames). Each time a function call is made, a new stack frame is constructed with the location of the call in the program, the arguments of the call, and the local variables of the function being called. To find out how a program got where it is, gdb provides the *backtrace* operation, which prints out one line per frame for all frames on the stack, starting from the current frame. Optionally, the *full* option can be passed to also print values of local variables. A frame can be selected with the *frame* command. With the *info* command, the user can print out a detailed description of the frame, including:

- The address of the frame.
- The address of the next frame down (called by this frame).
- The address of the next frame up (caller of this frame).
- The language in which the source code corresponding to this frame is written.
- The address of the frame's arguments.
- The address of the frame's local variables.
- The program counter saved in it (the address of execution in the caller frame).
- The registers that were saved in the frame.

The *info args* and *info locals* commands print information on all arguments and local variables of the selected frame, respectively.

Using the *print* command, an expression is evaluated and its value printed in a format appropriate to its data type.

Altering the State

To change the value of a variable, gdb provides the *set var* command. For example, *set var foo=5* would assign the value 5 to the variable foo (only if foo exists and 5 is a value that can be assigned to it).

Manually changing the instruction pointer is done with the *jump* command. It allows to resume execution at a specific location of the program.

State can be altered indirectly by using the *signal* command which sends a particular signal (specified by name or number) to the program.

To return manually from a function, the *return* command can be used, optionally specifying a return value.

Altering the Program

Programs run by gdb can be altered in several ways.

With the *set write on* command, the user can write to the loaded binary and patch it. This can be useful for emergency repairs.

Newer versions of gdb also support on-demand compilation and injection of code. This is achieved using the *compile code* command, whose argument is a valid line of source code. The compiled code is executed immediately and removed when finished.

Record/Replay

With the record/replay functionality of gdb, users can record the execution of a program and later replay it. Central to this process is the *record* command, which starts the *record and replay* process. By default, it fully captures the execution trace and allows replaying and reverse execution. While in this process, if the execution log includes the record for the next instruction, gdb will not really execute the instruction, but instead take all events that would happen normally from the execution log. If the execution log does not include a record for the next instruction, it is executed normally and its result recorded for future replay. The execution log can be traversed freely, and offers more control than the usual stepping functions. The *record goto* command allows the user to go to a specific location in the execution log. The *record delete* command deletes the subsequent execution log (in replay mode) and starts recording a new one. Record/replay is stopped with the *record stop* command.

Chosen Operations

We do not implement the full set of debugging operations offered by gdb, as we only want to prove feasibility. Only forward operations are implemented: stepping backwards in simulation or execution traces is covered in the next chapter. Similarly, we do not allow on-demand compilation of code or patching of binaries. This technique, called *live programming*, is also discussed in the next chapter. What remains are the following operations:

- *attach* the debugger to a running task, enabling other debugging operations;
- *detach* the debugger from a running task, disabling other debugging operations;
- *execute* a task as-fast-as-possible;
- *pause* a running task;
- execute a *line step*, which executes the task until the next source line in the currently executing frame is reached (similar to gdb's *next* command);
- execute a *big step*, which executes one rule (not present in gdb);
- execute a *small step*, which executes one phase of a rule (similar to gdb's *stepi* command);
- manage breakpoints: set a breakpoint at a source code line, optionally accompanied by a state condition, remove a breakpoint, list all breakpoints, and toggle a breakpoint (similar to gdb's breakpoint facilities).
- read the values of all symbols in the current scope (similar to gdb's state inspection facilities).

The next subsection explains how those operations are implemented.

5.1.3 De- and Reconstructed Simulator

Figure 5.1 shows our executor, instrumented with debugging support. Our goal was to support the operations presented in the previous subsection, while retaining the original execution semantics implemented in the Modelverse. This means we had to introduce points at which the algorithm can be interrupted (through stepping, breakpoints, etc.), as well as provide information on the state to the user of the debugger. Important with this formalism was to retain the possibility for users to interact with the running program that is being debugged. To implement the set of debugging operations, we first introduced new functions in the core algorithm of the Modelverse. We expanded the amount of debug information that was stored: instead of only storing the currently executing line, we store a complete call stack. When a function is called, a new item is pushed to the stack, and when a function returns, the top-most entry of the call stack is removed. We also added a function for retrieving the currently executing rule, and the currently executing phase.

The `execution_flow` component is responsible for executing rules, and reading the debug info associated with the execution. If the execution is running, it checks whether a rule needs to be executed. In case there is no rule to be executed, it might be the case that the program is waiting for input, so the executor goes to a timeout state that will recheck the condition after one second has passed, or when input is received. In case there is a rule to be executed, it executes one phase of a rule: a small step. This differs from the original Modelverse execution loop presented in Algorithm 2, where for each task, a number of rules are executed without interruption. For debugging purposes, this is no longer feasible:

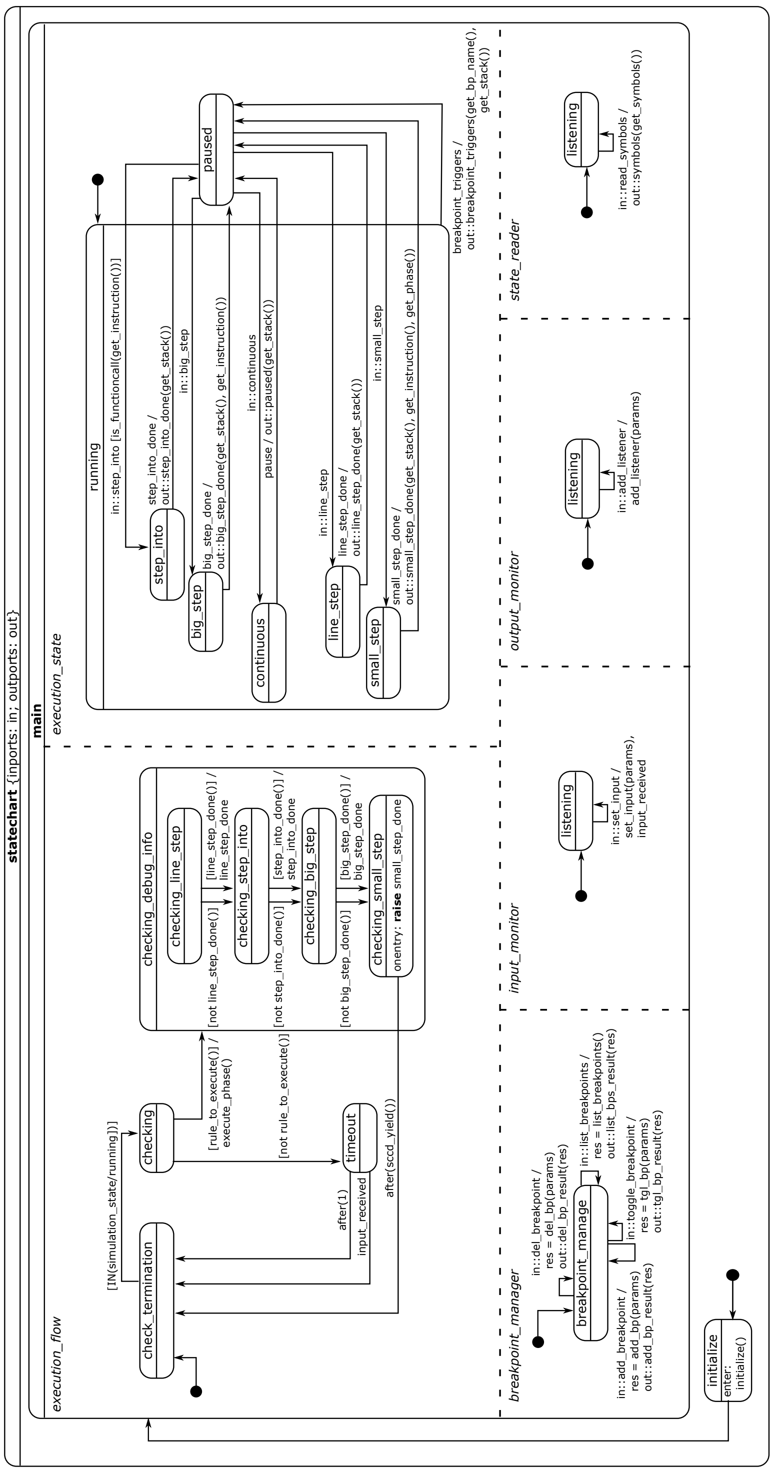

statechart {inports: in; outports: out}
main
execution_state
running
in::step_into [is_functioncall(get_instruction())]
step_into_done / out::step_into_done(get_stack())
in::big_step
big_step_done / out::big_step_done(get_stack(), get_instruction())
in::continuous
pause / out::paused(get_stack())
in::line_step
line_step_done / out::line_step_done(get_stack())
in::small_step
small_step_done / out::small_step_done(get_stack(), get_instruction(), get_phase())
breakpoint_triggers / out::breakpoint_triggers(get_bp_name(), get_stack())
paused
step_into
big_step
continuous
line_step
small_step
execution_flow
check_termination
[IN(simulation_state/running)]
checking
[rule_to_execute()] / execute_phase()
[not rule_to_execute()]
timeout
after(1)
input_received
after(sccd_yield())
checking_debug_info
checking_line_step
[line_step_done()] / line_step_done
[not line_step_done()]
checking_step_into
[step_into_done()] / step_into_done
[not step_into_done()]
checking_big_step
[big_step_done()] / big_step_done
[not big_step_done()]
checking_small_step
onentry: raise small_step_done
breakpoint_manager
breakpoint_manage
in::add_breakpoint / res = add_bp(params) out::add_bp_result(res)
in::del_breakpoint / res = del_bp(params) out::del_bp_result(res)
in::list_breakpoints / res = list_breakpoints() out::list_bps_result(res)
in::toggle_breakpoint / res = tgl_bp(params) out::tgl_bp_result(res)
input_monitor
listening
in::set_input / set_input(params), input_received
output_monitor
listening
in::add_listener / add_listener(params)
state_reader
listening
in::read_symbols / out::symbols(get_symbols())
initialize
enter: initialize()

Figure 5.1: The de- and reconstructed action language executor.

by attaching a debugger to a process, it is taken out of the "normal" execution loop of the Modelverse, and only one small step (rule) is executed each iteration. This of course decreases efficiency, but that is acceptable in a debugging context. After executing a rule, the Statecharts model checks for each type of step (*line step*, *step into*, *big step*, *small step*) whether one was completed. For example, we might have reached the next line by executing a rule, which means a line step was completed. In that case, a *line_step_done* event is raised. This event can trigger a transition in the `execution_state` component: if its active child state is `line_step`, the state is switched to `paused`, which means that the `execution_flow` will wait until the `execution_state` changes. Similar behaviour is implemented for *step into*, *big step*, and *small step*.

The `execution_state` keeps track of the execution state of a task. The task can be in three main states: `running`, `paused`, and `stopped`. The `running` state distinguishes between the mode selected by the user: either the user has requested to run the task continuously, or he requested a step. In case a step was requested, the system transitions to the `paused` state whenever that step type has finished. A `big step`, `small step`, and `line step` can always be executed. A `step into`, however, is only allowed if the next instruction is a function call. In `continuous` mode, the execution can be interrupted by a `pause` command from the user. From any `running` state, if a breakpoint triggers, the execution is paused and the user is notified.

We have four more parallel components:

- `breakpoint_manager` is responsible for adding, removing, toggling and listing breakpoints.
- `input_manager` allows a user to send input to the debugged task.
- `output_manager` allows a user to attach an output listener to the task.
- `state_reader` accepts requests for reading the current variables and their values.

The following functions were implemented as the non-modal behaviour of the debugger:

- *rule_to_execute()* is a Modelverse function that returns whether or not a rule needs to be executed; in case the task is waiting for input, this method returns false.
- *execute_phase()* is a Modelverse function that executes a single phase of a rule.
- *line_step_done()* is a function of the Statecharts model that checks whether a line step has ended. It does this by reading the current debugging info and comparing to the one saved when the user requested a line step. If the frame is the same, but the line has incremented, or if the frame was removed, a line step was executed.
- *step_into_done()* is a function of the Statecharts model that checks whether a step into has ended. It does this by reading the current debugging info and comparing to the one saved when the user requested a step into. If a frame was added, or the line has incremented, the step into was executed.
- *big_step_done()* is a function of the Statecharts model that checks whether a big step has ended. It does this by checking the current phase: if the phase is *finished*, the last small step of the big step was executed.

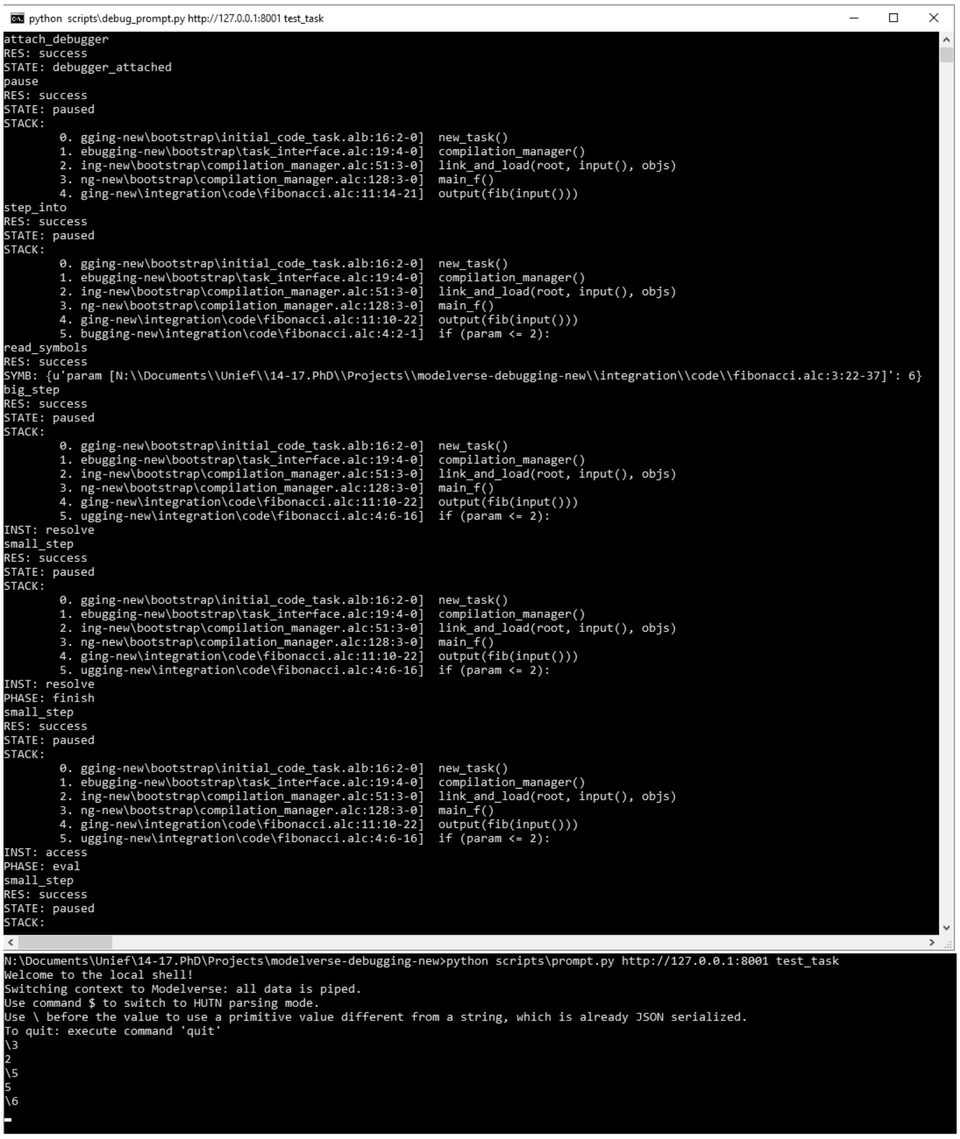
- *get_stack()* is a Modelverse function that returns the current stack of frames.
- *get_instruction()* is a Modelverse function that returns the currently executing instruction.
- *get_phase()* is a Modelverse function that returns the currently executing phase.
- *is_functioncall()* returns *True* if the passed instruction is a function call, otherwise it returns *False*.
- *breakpoint_triggers()* is a function of the Statecharts model that returns the breakpoint name if it triggers based on the debug info (the currently executing line) and the optional condition on the state.
- *get_symbols()* is a Modelverse function that returns all symbol names in scope and their values.

Summary We demonstrated in this subsection that the debugging operations described in the previous subsection have been successfully added to the execution algorithm of the Modelverse. In the next subsection, we couple the instrumented executor to a debugging environment.

5.1.4 Debugging Environment

Figure 5.2 presents the user interface for the action language debugger. On the bottom, the code implementing the Fibonacci function shown in Listing 5.1 is executed. Input is passed to the function by typing a literal integer value (prepended by a backslash). The function sends back the n^{th} Fibonacci number, where n was the input of the user, and the console application displays this output. In the window above, a debugger session is started. The debugger has been attached to the Fibonacci process by executing the *attach_debugger* operation. It was then paused, and the stack pointer is at line 11, blocking on the $input()$ function. The input 4 is passed to the program in the bottom-most window, but no reply is yet received, as the program execution is paused. The line contains three function calls. The user issues a *step_into* command and the program steps to the first line in a recursive call to the Fibonacci function. It does not step into the *input()* function, as one might expect, because this is a built-in function of the Modelverse. The *read_symbols* command returns all symbols that can be accessed in the current scope. In this case, the *param* variable has the value 6. A *big_step* command executes until the next instruction is reached, a *resolve* instruction to resolve the *param* variable. A *small_step* command executes one rule relating to the *resolve* instruction. The last executed *small_step* executes an access instruction to retrieve the value of the $params$ variable.

This debugging environment presents the debugging information at the most appropriate level of abstraction, using the abstractions the user is familiar with. Users can issue debugging commands by typing them, similarly to gdb. An extension of the debugger could integrate with a textual editor to visually indicate the currently executing line, for example, and offer a toolbar with buttons for each debugging operation. Such an interface is out of scope for this work, as we have proven feasibility.



```
python  scripts\debug_prompt.py http://127.0.0.1:8001 test_task
```

```
attach_debugger
RES: success
STATE: debugger_attached
pause
RES: success
STATE: paused
STACK:
        0. gging-new\bootstrap\initial_code_task.alb:16:2-0]  new_task()
        1. ebugging-new\bootstrap\task_interface.alc:19:4-0]  compilation_manager()
        2. ing-new\bootstrap\compilation_manager.alc:51:3-0]  link_and_load(root, input(), objs)
        3. ng-new\bootstrap\compilation_manager.alc:128:3-0]  main_f()
        4. ging-new\integration\code\fibonacci.alc:11:14-21]  output(fib(input()))
step_into
RES: success
STATE: paused
STACK:
        0. gging-new\bootstrap\initial_code_task.alb:16:2-0]  new_task()
        1. ebugging-new\bootstrap\task_interface.alc:19:4-0]  compilation_manager()
        2. ing-new\bootstrap\compilation_manager.alc:51:3-0]  link_and_load(root, input(), objs)
        3. ng-new\bootstrap\compilation_manager.alc:128:3-0]  main_f()
        4. ging-new\integration\code\fibonacci.alc:11:10-22]  output(fib(input()))
        5. bugging-new\integration\code\fibonacci.alc:4:2-1]  if (param <= 2):
read_symbols
RES: success
SYMB: {u'param [N:\\Documents\\Unief\\14-17.PhD\\Projects\\modelverse-debugging-new\\integration\\code\\fibonacci.alc:3:22-37]': 6}
big_step
RES: success
STATE: paused
STACK:
        0. gging-new\bootstrap\initial_code_task.alb:16:2-0]  new_task()
        1. ebugging-new\bootstrap\task_interface.alc:19:4-0]  compilation_manager()
        2. ing-new\bootstrap\compilation_manager.alc:51:3-0]  link_and_load(root, input(), objs)
        3. ng-new\bootstrap\compilation_manager.alc:128:3-0]  main_f()
        4. ging-new\integration\code\fibonacci.alc:11:10-22]  output(fib(input()))
        5. ugging-new\integration\code\fibonacci.alc:4:6-16]  if (param <= 2):
INST: resolve
small_step
RES: success
STATE: paused
STACK:
        0. gging-new\bootstrap\initial_code_task.alb:16:2-0]  new_task()
        1. ebugging-new\bootstrap\task_interface.alc:19:4-0]  compilation_manager()
        2. ing-new\bootstrap\compilation_manager.alc:51:3-0]  link_and_load(root, input(), objs)
        3. ng-new\bootstrap\compilation_manager.alc:128:3-0]  main_f()
        4. ging-new\integration\code\fibonacci.alc:11:10-22]  output(fib(input()))
        5. ugging-new\integration\code\fibonacci.alc:4:6-16]  if (param <= 2):
INST: resolve
PHASE: finish
small_step
RES: success
STATE: paused
STACK:
        0. gging-new\bootstrap\initial_code_task.alb:16:2-0]  new_task()
        1. ebugging-new\bootstrap\task_interface.alc:19:4-0]  compilation_manager()
        2. ing-new\bootstrap\compilation_manager.alc:51:3-0]  link_and_load(root, input(), objs)
        3. ng-new\bootstrap\compilation_manager.alc:128:3-0]  main_f()
        4. ging-new\integration\code\fibonacci.alc:11:10-22]  output(fib(input()))
        5. ugging-new\integration\code\fibonacci.alc:4:6-16]  if (param <= 2):
INST: access
PHASE: eval
small_step
RES: success
STATE: paused
STACK:
```

```
N:\Documents\Unief\14-17.PhD\Projects\modelverse-debugging-new>python scripts\prompt.py http://127.0.0.1:8001 test_task
Welcome to the local shell!
Switching context to Modelverse: all data is piped.
Use command $ to switch to HUTN parsing mode.
Use \ before the value to use a primitive value different from a string, which is already JSON serialized.
To quit: execute command 'quit'
\3
2
\5
5
\6
```

Figure 5.2: The interface for the action language debugger.

Summary

This section demonstrates how debugging support can be implemented for an action language. The action language exhibits the following semantic features of the feature diagram presented in Figure 4.2:

- It is **procedural** (functions can call other functions).
- It allows for **environment interaction** (through input/output functions).
- It is **untimed**.
- It is **deterministic**.
- It has **static structure**.

The first two features are the most prominent for this formalism, and in this section, we have demonstrated that our approach can support debugging for languages exhibiting these features.

5.2 Causal Block Diagrams

Causal Block Diagrams (CBD) [33], also known as Synchronous Data Flow, is a visual modelling language that provides abstractions to model mathematical expressions. It is a causal, timed (continuous or discrete), static-structure, deterministic formalism that allows for algebraic loops to be modelled.

5.2.1 Syntax and Semantics

The CBD language offers abstractions to model dataflow: data flows between elements over time and is manipulated. A model in the design language of CBD consists of:

- blocks that model mathematical operators, expressions that evaluate to Boolean values, memory, and constants;
- connectors between blocks: each block has one outgoing connector, and zero or more incoming connectors.

Connecting blocks results in a network that can contain algebraic loops.

There are three different notions of time in CBDs:

1. Algebraic models, which do not have a notion of time, and are only capable of modelling basic mathematical equations.
2. Discrete-time models, where time is updated by a fixed amount after each evaluation of all blocks. The *delay* block introduces memory by providing the previous value of its input signal on its output.

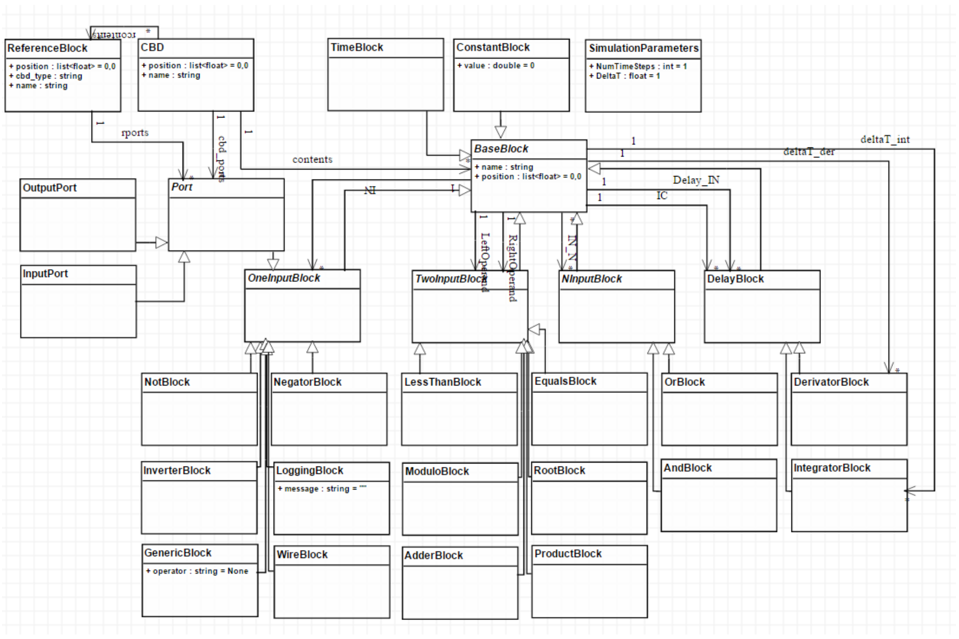


Figure 5.3: The metamodel of the CBD language.

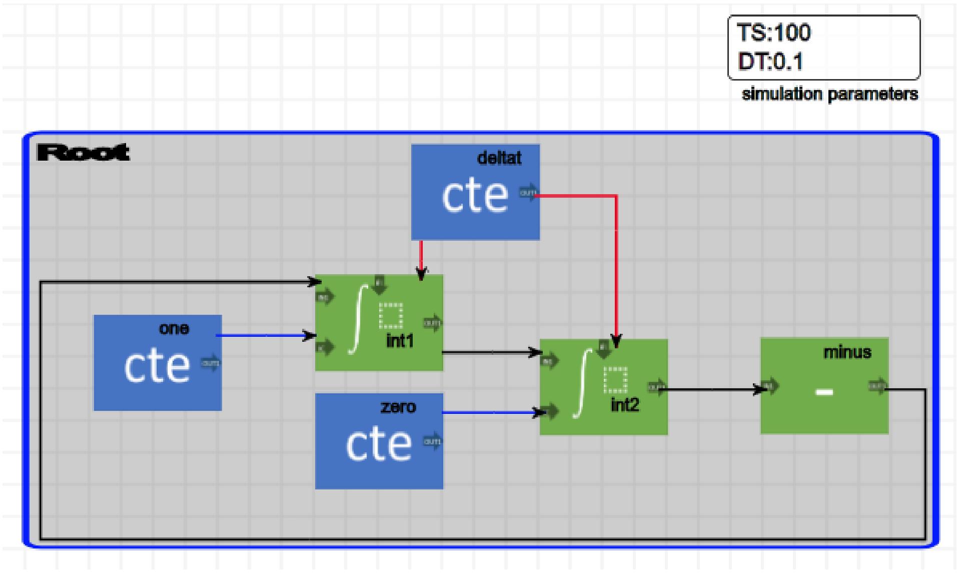
3. Continuous-time models, which can only be simulated after approximation. Time is updated with a sufficiently small δ_t value, turning the model into a discrete-time model where the state of the system is sampled sufficiently often to approach a continuous function. The *derivative* and *integrator* blocks are added and their results approach a continuous function.

An example continuous-time CBD model is shown in Figure 5.4a. It models a "harmonic oscillator", a model typically used to gauge the effect of simulation parameters on simulation error. The following equations are modelled:

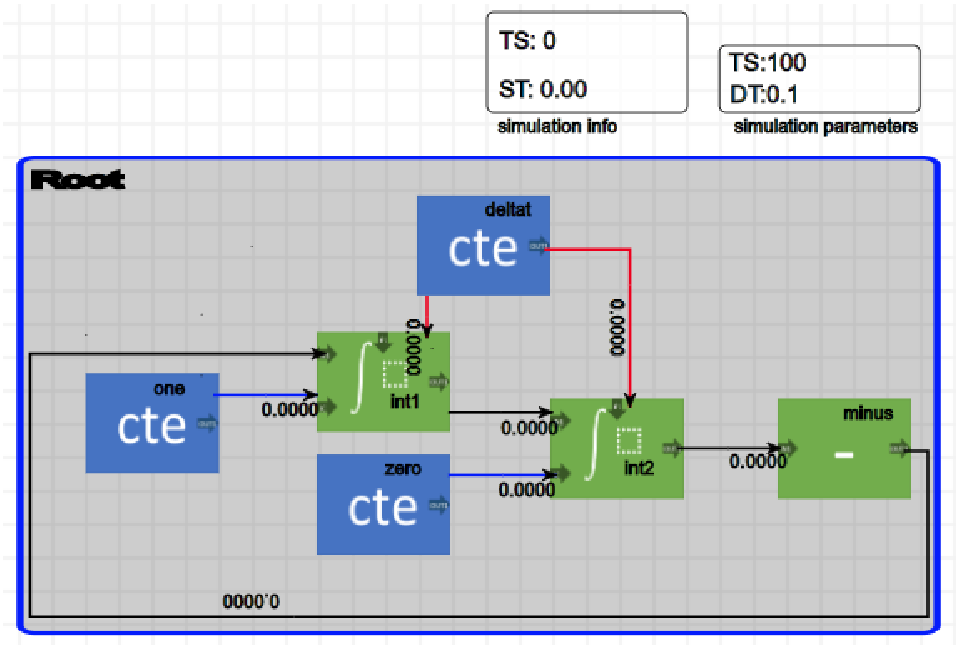
$$\begin{cases} \frac{d^2x}{dt^2}(t) = -x(t) \\ \frac{dx}{dt}(0) = 1 \\ x(0) = 0 \end{cases} \tag{5.1}$$

From left to right, the first integrator block receives as input the signal of $\frac{d^2x}{dt^2}(t)$. It outputs the value of $\frac{dx}{dt}$. Its initial condition is 1, which corresponds to the equation $\frac{dx}{dt}(0) = 1$. The second integrator block receives as input the signal of $\frac{dx}{dt}(t)$. It outputs the value of x. Its initial condition is 0, which corresponds to the equation $x(0) = 0$. Last, a loop is created to model the equation $\frac{d^2x}{dt^2}(t) = -x(t)$. This loop can be algebraic, depending on the integration method used, which requires it to be passed to a linear solver.

The design model also contains the simulation parameters, which specify the step size δ_t to be used, and the number of simulation iterations. In our example, $\delta_t = 0.1$ and 100



(a) A CBD model in a design environment.



(b) The CBD model initialized in a simulation environment.

Figure 5.4: An example CBD model, both its design and initialized runtime state.

simulation iterations are to be executed. At the end of simulation, the simulated time will be 10 ($= 100 * 0.1$). At runtime, the connections between blocks carry *signals*: values as

ALGORITHM 3: The CBD simulator's "main loop".

```
Input: Model to simulate ($model$)
Output: Final state of the simulation and the simulated time
$time \leftarrow 0$;
$state \leftarrow initialize\_state(model)$;
while not end_condition(time, state) do
    $schedule \leftarrow loop\_detect(dep\_graph(model))$;
    for block in schedule do
        $compute\_block(state, block)$;
    end
    $time \leftarrow time + \delta_t$;
end
return $get\_state(), get\_time()$
```

function over time. A CBD is simulated by updating the values of the outgoing signals of all blocks each iteration. The pseudocode for the algorithm of the CBD simulator is shown in Algorithm 3.

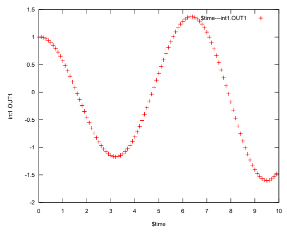
Four functions are central to the algorithm:

- *loop_detect* computes all loops found in the CBD. A loop exists when the input of a block is computed by another block reachable from the original block. The function returns a collection of all blocks and loops, in the order that they need to be computed. Only *linear* loops can be solved: if the model describes a nonlinear system, a nonlinear solver would have to be used, which the simulator used does not support.

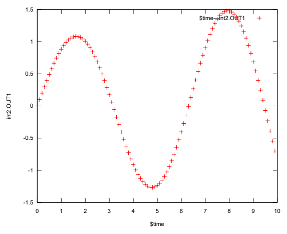
- *dep_graph* computes dependencies between blocks, which is needed for the loop detection algorithm.

- *compute_block* contains the code that performs each block's computation on its input signals, updating the value of its outgoing signals (for example, the current value of the output signal of a sum block equals the sum of the current values of its input signals).

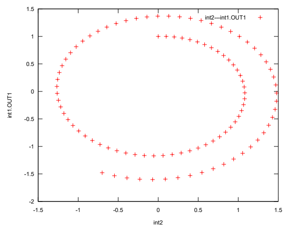
Each time step, the simulator iterates over all blocks in the correct order and computes the values of each block's outgoing signals. The end condition depends on the simulated time. It is modelled in the "simulation parameters" element in the graphical user interface (see Figure 5.4). Here, the simulation will end after 100 time steps. The time increment (δ_t) is set to 0.1. The model and parameters are compiled to the format the simulator requires and is then initialized by the simulator. The initial state of the model can be displayed in an appropriate runtime environment, as seen in Figure 5.4b. The runtime environment adds a *simulation info* element, which contains the current step TS and the current simulated time ST. When the simulator completes a run, it returns the final state, which is displayed in the user interface. A mapping between elements in the user interface model and the compiled model (and back) realizes this visualization.



(a) The signal values of the first integrator, which approximates $\frac{dx}{dt}(t)$.



(b) The signal values of the second integrator, which approximates x.



(c) Plotting $\frac{dx}{dt}(t)$ versus x to estimate simulation error.

Figure 5.5: Simulation results of the example CBD model.

The result of simulation is a set of time-indexed signal values for each block. Figure 5.5 plots the signal values of the first and second integrator. From the figure, we see that $\frac{dx}{dt}$ approximates the cosine function, and x approximates the sine function. The third plot is a parametric curve, where $\frac{dx}{dt}(t)$ and x are plotted. Since these functions approximate the cosine and sine function, the curve will approximate a circle with radius one (since $cos^2(x) + sin^2(x) = 1$). But, with simulation error accumulating over time, the circle will start to spiral, as can be seen from the figure.

5.2.2 Debugging Operations

We define the following debugging operations for the CBD formalism:

- **[Continuous Simulation]** runs the simulation until completion, and communicates the final state of the simulation when finished.
- **[Realtime Simulation]** runs the simulation until completion, but synchronizes the simulated time with the wall-clock time. The state is shown each time step, such that the modeller can see how the system evolves over time. The speed depends on the δ_t parameter, but a scale factor can be applied to slow down or speed up simulation.
- **[Pause]** interrupts a running simulation. In continuous simulation, the simulation stops after the currently executing "big step" has finished, ensuring the system is in a stable state. In realtime simulation, however, the simulation is paused as soon as possible. This means that the simulator can be "in between" two big steps, as it might have been waiting for the appropriate wall-clock time to elapse when the pause was requested.
- **[Big Step]** advances the simulation with one "big step" (one iteration of the outer *while*-loop) and communicates the state of the simulation.
- **[Small Step]** advances the simulation with one "small step" (one iteration of the inner *while*-loop) and communicates the state of the simulation. This allows the user to step through individual block computations.
- **[Reset]** allows the user to reset the simulation to the initial state.

- **[Breakpoints]** can be modelled as elements that read the output value of a block in the model. The breakpoint pauses the simulation when the output of its connected element has a non-zero value. This allows users to model arbitrary conditions on which the simulation needs to pause (as they can be modelled as a network of CBD blocks) in the CBD model.
- If nonlinear (unsolvable) components are found, the debugger indicates which blocks belong to that component, in order for the modeller to be notified so appropriate changes can be made to the model.

5.2.3 De- and Reconstructed Simulator

Debugging a CBD simulation requires "lifting out" the outer *while*-loop, to implement "big step" behaviour. On a more fine-grained level, the inner loop can be lifted out too, to implement "small step" behaviour by exposing the computation of individual blocks.

Figure 5.6 presents the Statecharts model resulting from de- and reconstructing the CBD simulation algorithm. Our goal was to support the operations presented in the previous subsection, while retaining the original execution semantics implemented by the simulator. This means we had to introduce points at which the algorithm can be interrupted (through stepping, breakpoints, etc.), as well as provide information on the state to the user of the debugger. Important in this formalism is the distinction between "big steps" (an iteration of the *while*-loop of the simulator) and "small steps" (the computation of a single signal value). The Statecharts model consists of four orthogonal components: `simulation_state`, `simulation_flow`, `reset_monitor` and `breakpoint_manager`.

The `simulation_state` orthogonal component determines in what execution mode the simulator is. It describes how the user can influence the control flow of the simulation. Two top-level states are defined: `paused` and `running`. The `running` state has three substates, which encode the simulation modes: `continuous`, `realtime`, `big_step`, and `small_step`. The simulator starts in the `paused` state, and can only transition to one of the substates of `running` when a user event is received. Pausing is only possible when running the simulation in continuous mode or in realtime mode. The debugging-enhanced simulator does not allow pausing the simulation when computing a big step or small step, as that could leave the simulator in a (globally) inconsistent state. Rather, the simulator pauses automatically after every big step when in big step mode, and after every small step in small step mode. When the simulation terminates, the simulation state transitions to the `paused` state from any of the `running` states. Detecting termination is encoded in the `simulation_flow` orthogonal component, discussed below. Based on the simulation state, the behaviour of the simulation flow will change.

The `simulation_flow` orthogonal component determines the flow of a single simulation step, and explicitly encodes the invocation order of the non-modal functions. This is an explicit representation of the control flow that was implemented in the `while`-loop in the coded implementation. The orthogonal component consists of four states: `check_termination`, `waiting`, `checking`, and `checking_step`. In the `check_termination` state, the simulator checks whether or not a next step should be computed. This check inspects the state of the orthogonal component `simulation_state`. There are three possibilities:

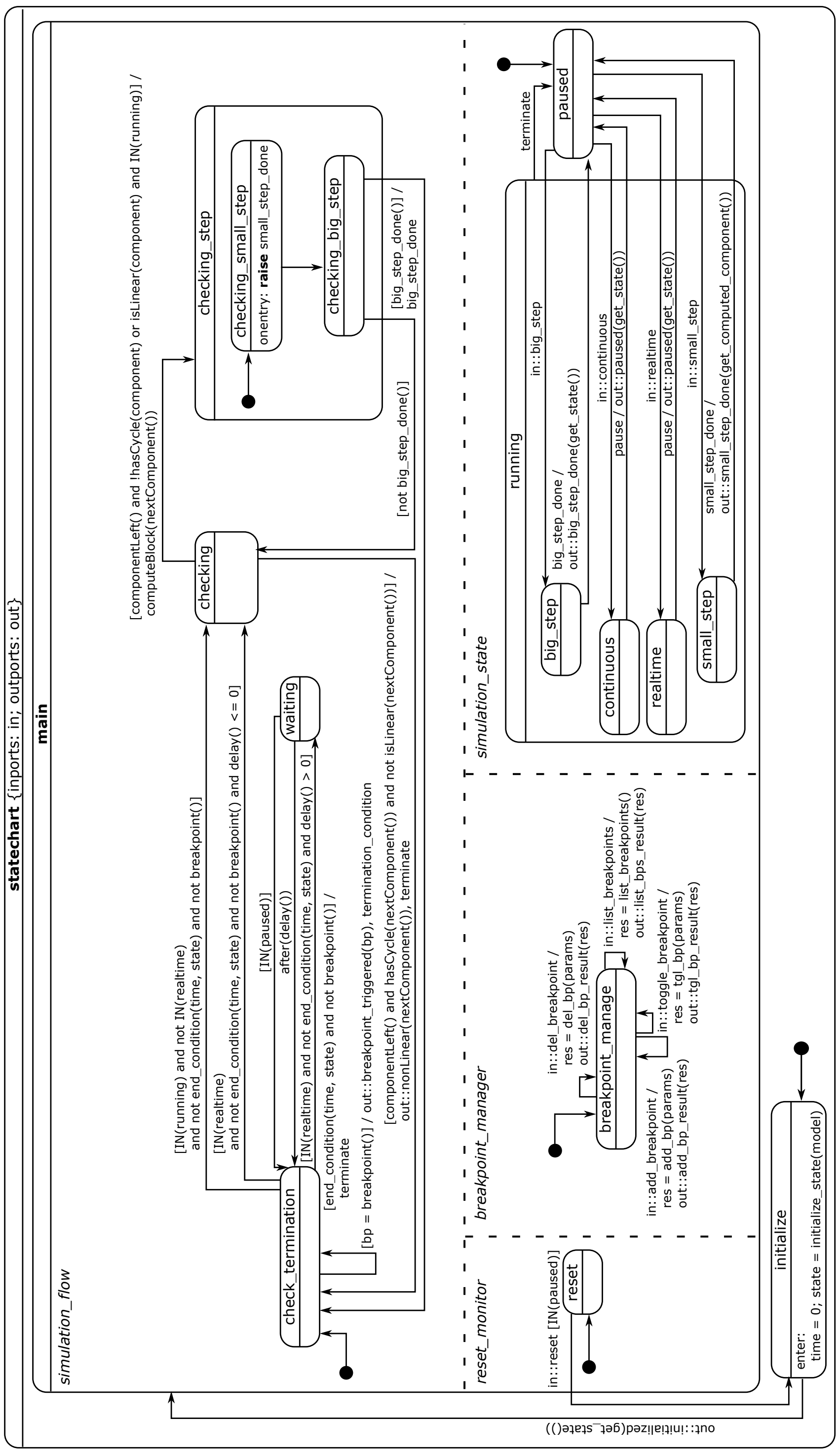

statechart {inports: in; outports: out}
main
simulation_flow
check_termination
checking
checking_step
checking_small_step
onentry: raise small_step_done
checking_big_step
waiting
[IN(running) and not IN(realtime) and not end_condition(time, state) and not breakpoint()]
[IN(realtime) and not end_condition(time, state) and not breakpoint() and delay() <= 0]
[IN(paused)]
after(delay())
[IN(realtime) and not end_condition(time, state) and delay() > 0]
[end_condition(time, state) and not breakpoint()] / terminate
[bp = breakpoint()] / out::breakpoint_triggered(bp), termination_condition
[componentLeft() and hasCycle(nextComponent()) and not isLinear(nextComponent())] / out::nonLinear(nextComponent()), terminate
[componentLeft() and !hasCycle(component) or isLinear(component) and IN(running)] / computeBlock(nextComponent())
[big_step_done()] / big_step_done
[not big_step_done()]
simulation_state
paused
terminate
running
big_step
continuous
realtime
small_step
in::big_step
big_step_done / out::big_step_done(get_state())
in::continuous
pause / out::paused(get_state())
in::realtime
pause / out::paused(get_state())
in::small_step
small_step_done / out::small_step_done(get_computed_component())
breakpoint_manager
breakpoint_manage
in::del_breakpoint / res = del_bp(params) out::del_bp_result(res)
in::list_breakpoints / res = list_breakpoints() out::list_bps_result(res)
in::toggle_breakpoint / res = tgl_bp(params) out::tgl_bp_result(res)
in::add_breakpoint / res = add_bp(params) out::add_bp_result(res)
reset_monitor
reset
in::reset [IN(paused)]
initialize
enter: time = 0; state = initialize_state(model)
out::initialized(get_state())

Figure 5.6: The de- and reconstructed CBD simulator.

1. The simulator is running, but not in real-time. In that case, the next simulation step can be computed.
2. The simulator is running in real-time. In that case, either the delay between two simulation steps has passed, and the next simulation step can be computed. Or, the delay has not passed, and the simulator has to wait. The simulator transitions to the `waiting` state, and a transition is scheduled to fire after the delay has passed. But, pausing needs to be possible while waiting. So, a transition is added which lets the control go back to the `check_termination` state when the user requests a pause.
3. A breakpoint triggers, in which case the simulation should pause.

The `checking` state is responsible for checking the next component: if it is nonlinear, an event is raised to the user and the simulation is paused. Otherwise, the output of the next block is computed. In `checking_step`, a *small_step_done* event is always raised, since a small step corresponds to the computing of one block. A *big_step_done* event is raised if the last component of the current time step is computed. These events can influence the `simulation_state` orthogonal component. In the *checking* and *check_termination* states, the state of that orthogonal component is always checked to ensure the next simulation step can be computed.

The `reset_monitor` listens for user requests to reset the simulation, and allows it when the simulation is paused.

The `breakpoint_manager` can add, toggle, and delete breakpoints, as well as list all currently defined breakpoints.

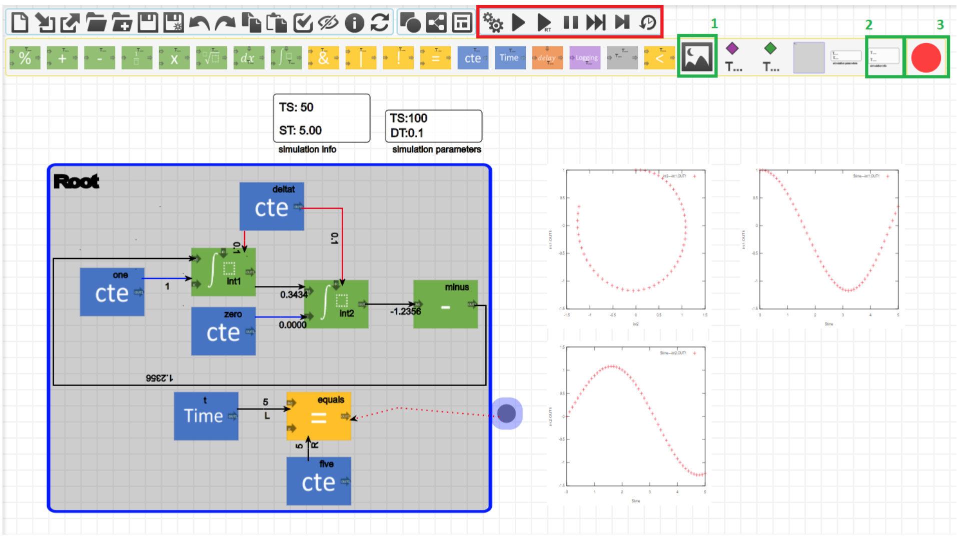
Summary We demonstrated in this subsection that the debugging operations described in the previous subsection have been successfully added to the CBD simulator. In the next subsection, we couple the instrumented executor to a debugging environment.

5.2.4 Debugging Environment

The debugging environment for the CBD formalism is shown in Figure 5.7. A toolbar allows the user to access the functions of the debugging-enhanced simulator: it is marked in red. The current state of the simulation (an instance of the runtime formalism) is displayed on the canvas. For each signal (connection between blocks), its current value is displayed. The model shown is paused, since a breakpoint triggered (represented by the small black circle that is highlighted in blue). The breakpoint triggers when the simulated time equals 5.

To debug the model effectively, three elements were added as part of the debugging language:

1. A plotter, of which three instances are shown. This element draws the value of a signal of one of the blocks as it evolves over time. The plot is updated after every state update coming from the simulator. In realtime simulation, the state is communicated after every big step of the simulation algorithm. The state is also communicated when the simulation terminates or pauses. The plots show the evolution of the state variables up to the current point in simulated time.


TS: 50
ST: 5.00
simulation info

TS:100
DT:0.1
simulation parameters

Figure 5.7: The debugging environment for the CBD formalism.

2. A "simulation info" element, which is also instantiated on the canvas. It contains the current simulation step (*TS*) and the simulated time (*ST*). As can be seen from the figure, the simulation is currently halfway: the user has specified that 100 simulation steps are to be executed, the current simulation step is 50.

3. A breakpoint element, which is used to model a condition on the runtime state of the system at which the simulator has to automatically pause. The breakpoint can be placed on the canvas and connected to a block. It triggers when the output of that block is non-zero.

Summary

This section demonstrates how debugging support can be implemented for Causal Block Diagrams. The language exhibits the following semantic features of the feature diagram presented in Figure 4.2:

- It allows to model **algebraic loops** (that need a specific schedule to be computed in a consistent manner).
- It implements **discrete-time** or **continuous-time** semantics (depending on the types of blocks used in the instance models).
- It is **deterministic**.
- It has **static structure**.

The modelling of algebraic loops is the most prominent feature of this formalism, and in this section, we have demonstrated that our approach can support debugging for languages exhibiting this feature.

5.3 Parallel DEVS

Parallel DEVS [38] is an extension of DEVS [221], a popular formalism for modelling complex dynamic systems using a discrete-event abstraction. It is a causal, timed (discrete-event), static-structure, deterministic formalism that allows to model systems that interact with their environment through input and output functions.

5.3.1 Syntax and Semantics

The Parallel DEVS formalism consists of atomic models, which are connected in a coupled model. The atomic models are behavioural (they exclusively model behaviour), while the coupled models are structural (they exclusively model structure).

Atomic DEVS

An *atomic* DEVS model is a structure:

$$M =< X, S, Y, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$$

The *input set X* denotes the set of admissible inputs of the model. The input set may have multiple ports, denoted by m in this definition. X is a structured set

$$X = \times_{i=1}^{m} X_i$$

where each of the X_i denotes the admissible inputs on port i. The *output set Y* denotes the set of admissible outputs of the model. The output set may have multiple ports, denoted by l in this definition. Y is a structured set

$$Y = \times_{i=1}^{l} Y_i$$

where each of the Y_i denotes the admissible outputs on port i. The *state set S* is the set of admissible sequential states. Typically, S is a structured set

$$S = \times_{i=1}^{n} S_i$$

The *internal transition function* δ_{int} defines the next sequential state, depending on the current state. It is triggered after the time returned by the *time advance function* has passed (in the simulation, not in wall-clock time). Note that this function does not require the elapsed simulation time as an argument, since it will always be equal to the *time advance function*.

$$\delta_{int} : S \rightarrow S$$

The *output function* λ maps the sequential state set onto an output bag. Output events are only generated by a DEVS model at the time of an *internal/confluent transition*. This function is called *before* the transition function is called, so the state that is used will be the state before the transition happens.

$$\lambda : S \rightarrow Y^b$$

The *external transition function* δ_{ext} gets called whenever an *external input* ($\in X$) is received in the model. The signature of this transition function is

$$\delta_{ext} : Q \times X^b \rightarrow S$$
$$\text{with } Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$$

with *e* the elapsed time since the last transition.

When the *external transition function* is called, the *time advance function* is called again and the previously scheduled internal event is rescheduled with the new value. The *time advance function ta* defines the simulation time the system remains in the current state before triggering the *output functions* and *internal transition functions*.

$$ta : S \rightarrow \mathbb{R}^+_{0,+\infty}$$

Note that $+\infty$ is included, since it is possible for a model to *passivate* in a certain state, meaning that it will never have an internal transition in this state.

Should the internal and external transition function be called at exactly the same point in simulated time, the *confluent transition function* is called instead. It is defined as

$$\delta_{conf} : S \times X^b \rightarrow S$$

The behaviour of atomic DEVS models is summarized in Figure 5.8.

Coupled DEVS

A *coupled* DEVS model is a structure:

$$M =< X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$$

With X_{self} and Y_{self} the *input* and *output* set respectively.

D is the set of unique component references (names), the coupled model itself is not an element of D.

$\{M_i\}$ is the set of components containing the atomic DEVS structure of all subcomponents referenced by elements in D.

$$\{M_i | i \in D\}$$

The set of *influencees* $\{I_i\}$ determines the elements whose input ports are connected to output ports of component i. Note that $self$ is included in this definition, as a component can send (receive) messages to (from) the coupled model itself.

$$\{I_i | i \in D \cup \{self\}\}$$

A component cannot influence components outside of the current coupled model, nor can it influence itself directly as self-loops are forbidden.

$$\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$$
$$\forall i \in D \cup \{self\} : i \notin I_i$$

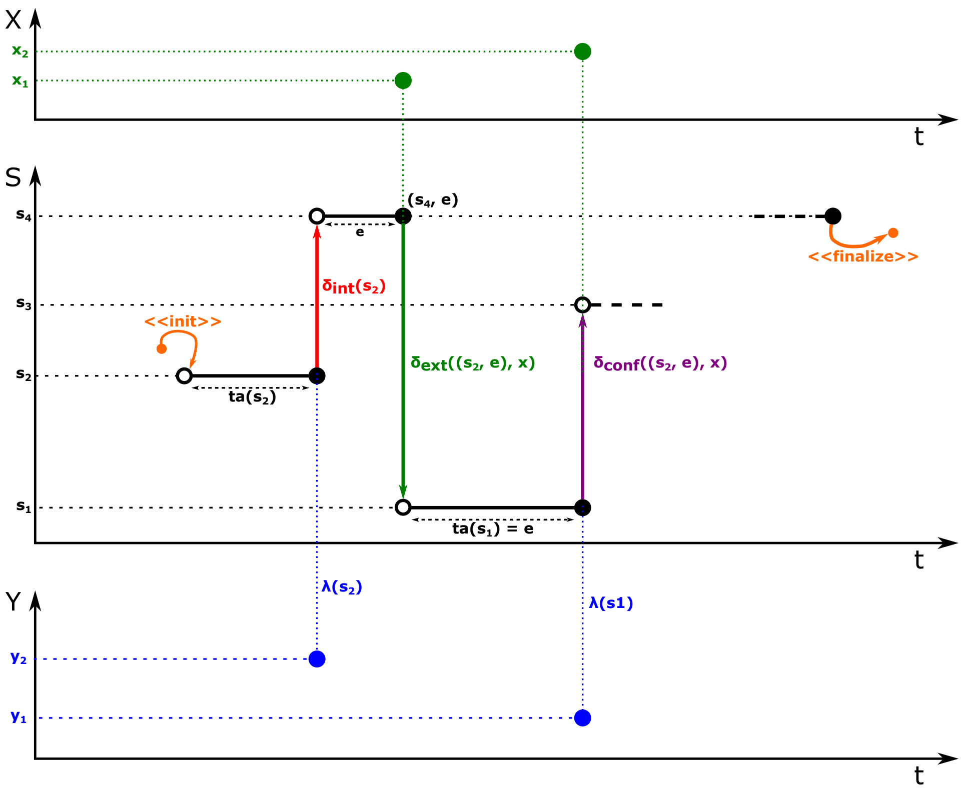


Figure 5.8: Input, state, and output trajectories for an atomic DEVS model.

The couplings are further specified by the *transfer functions* $Z_{i,j}$. These functions are applied to the messages being passed, depending on the output and input port. These functions allow for reuse, since they allow output events to be made compatible with the input set of the connected models.

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}$$

DEVS was originally not intended to be a visual language, instead it relies on its mathematical description in the form of sets and transitions that modify these sets. To debug DEVS models, however, a visual interface is helpful. We have developed such an interface, which allows users to model and simulate (and debug) DEVS models. We start by describing the graphical design environment. Figure 5.9 presents the metamodel for this Parallel DEVS design formalism. It consists of the following elements:

- **BaseDEVS** is the superclass for the Coupled DEVS and Atomic DEVS classes. It has the following attributes:
  - **name** is the name of the model. It can be used in DEVS instances (see below) to refer to their type.
  - **attributes** is a list of attributes for this model.

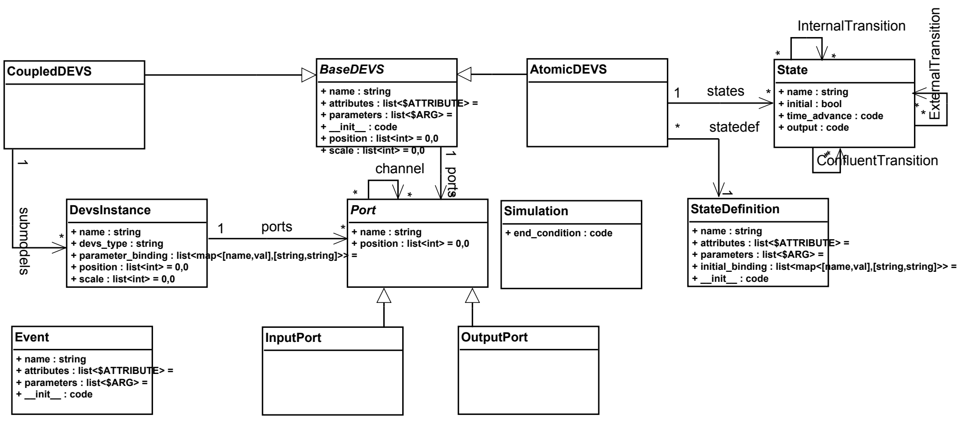

CoupledDEVS
BaseDEVS
+ name : string
+ attributes : list<$ATTRIBUTE> =
+ parameters : list<$ARG> =
+ __init__ : code
+ position : list<int> = 0,0
+ scale : list<int> = 0,0
AtomicDEVS
State
+ name : string
+ initial : bool
+ time_advance : code
+ output : code
InternalTransition
ExternalTransition
ConfluentTransition
states
statedef
channel
ports
submodels
DevsInstance
+ name : string
+ devs_type : string
+ parameter_binding : list<map<[name,val],[string,string]>> =
+ position : list<int> = 0,0
+ scale : list<int> = 0,0
Port
+ name : string
+ position : list<int> = 0,0
Simulation
+ end_condition : code
StateDefinition
+ name : string
+ attributes : list<$ATTRIBUTE> =
+ parameters : list<$ARG> =
+ initial_binding : list<map<[name,val],[string,string]>> =
+ __init__ : code
Event
+ name : string
+ attributes : list<$ATTRIBUTE> =
+ parameters : list<$ARG> =
+ __init__ : code
InputPort
OutputPort

Figure 5.9: The Parallel DEVS metamodel in AToMPM.

- **parameters** is a list of parameters passed to the constructor of this model.
- **__init__** is the constructor for this model. The constructor is responsible for initializing the values of this model's attributes. To do this, the code can access the parameters by name.
- **position** and **scale** are visualization-specific attributes.

- **Port** is a class to represent the ports of DEVS models. There are two types of ports: input ports and output ports. Each port has a name. Ports can be connected to other ports through the **channel** association. Channels have one attribute: a transfer function that translates events when they are transferred through the channel.
- **CoupledDEVS** instances can contain a number of submodels (which are instances of other atomic/coupled DEVS models). Exactly one CoupledDEVS model should have the name **Root** and is the root model used for simulation.
- **DevsInstance** is an instantiation of an atomic or coupled DEVS model, its type. The type of a DevsInstance needs to be present in the same model. It has the following attributes:
  - **name** is the name of the instance. It has to be unique within the enclosing coupled DEVS model.
  - **devs_type** is a reference to the instance's type, an atomic or coupled DEVS model. The type needs to be defined in the same model as the reference.
  - **parameter_binding** is a mapping of parameter names onto values passed to the constructor of the DEVS model this is an instance of.
  - **position** and **scale** are visualization-specific attributes.
- **AtomicDEVS** elements contain a number of states, and are connected to exactly one state definition.

- **StateDefinition** models the "template" of a state for an atomic DEVS model. It has the following attributes:
  - **name** is the name of this state definition.
  - **attributes** is a list of attributes for this state definition. It is a list of "state variables". By default, all *StateDefinition* instances have one attribute: a *name*, whose value corresponds to the name of the current state of the atomic DEVS model.
  - **parameters** is a list of parameters for the constructor, which is called when a state change occurs.
  - **__init__** is the code for the constructor.
- **State** instances model the discrete states of an atomic DEVS model. It has the following attributes:
  - **name** is the name of the state.
  - **initial** denotes whether this is the initial state of the atomic DEVS model. There is exactly one initial state for each atomic DEVS model.
  - **time_advance** is the block of code which computes the time the atomic DEVS model should stay in this state before firing its internal transition. It should return a positive floating point number, or ∞.
  - **output** is the block of code which computes the output that is generated when a state transition occurs from this state. It should return a mapping between port names and a list of event instances.
- **InternalTransition** instances connect two states with each other, and specifies which state to transition to after the time advance of the source state has passed. It has two attributes:
  - **condition** allows to specify a condition which needs to evaluate to *True* in order for the internal transition to be executed. Two internal transitions leaving the same state should have non-overlapping conditions (to avoid non-determinism).
  - **action** allows to specify an action when performing the transition. The action can change the state attributes of the atomic DEVS model (that were defined in the *StateDefintition* instance associated with the atomic DEVS model).
- **ExternalTransition** instances connect two states with each other, and specifies which state to transition to after an input was received on an input port. It has two attributes:
  - **condition** allows to specify a condition which needs to evaluate to *True* in order for the internal transition to be executed. Two internal transitions leaving the same state should have non-overlapping conditions.
  - **action** allows to specify an action when performing the transition. The action can change the state attributes of the atomic DEVS model (that were defined in the *StateDefintition* instance associated with the atomic DEVS model). The action code can access an input dictionary called *inputs*, which maps all port

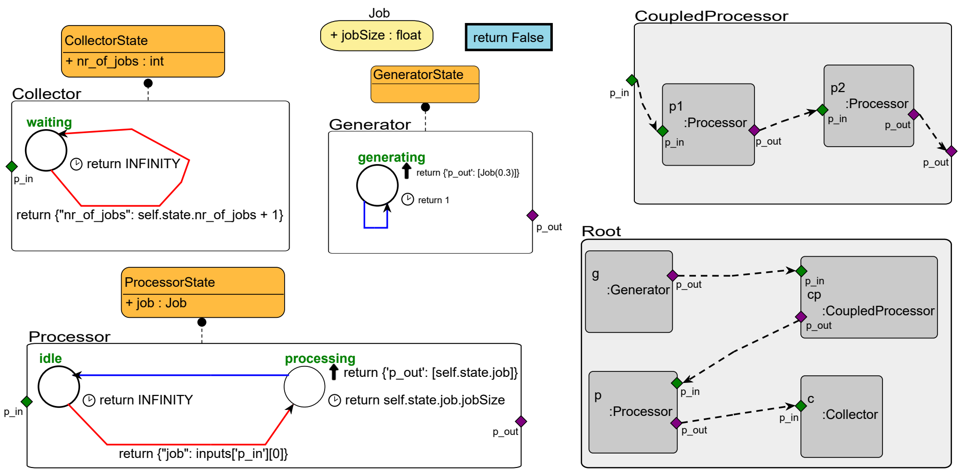


Figure 5.10: The design model of the producer-consumer DEVS model.

names onto the bag of inputs that was received on that port, and the variable *elapsed*, which contains the elapsed time since the last external transition.

- **ConfluentTransition** instances connect two states with each other, and specifies which state to transition to when there is a conflict between internal and external transition function. By default, the internal transition function is executed first, followed by the external transition. It has two attributes:
  - **condition** allows to specify a condition which needs to evaluate to *True* in order for the confluent transition to be executed. Two confluent transitions leaving the same state should have non-overlapping conditions.
  - **action** allows to specify an action when performing the transition. The action can change the state attributes of the atomic DEVS model (that were defined in the *StateDefintition* instance associated with the atomic DEVS model). The action code can access an input dictionary called *inputs*, which maps all port names onto the bag of inputs that was received on that port, and the variable *elapsed*, which contains the elapsed time since the last external transition.
- **Simulation** allows to specify an end condition for the simulation. The simulated time can be accessed with the variable *time*. The current state of the model can be accessed using the *model* variable.

In Figure 5.10, an example model specifying a producer-consumer system is shown in the generated AToMPM modelling environment. It consists of the following atomic DEVS models:

- **Generator** puts a new *Job* message on its output port every 0.3 simulated time seconds.
- **Collector** accepts a *Job* message on its input port and counts the number of messages it has received (by increasing its *nr_of_jobs* state variable).

- **Processor** accepts a *Job* on its input port, does some computation on it for a predetermined amount of time, and then puts the *Job* message on its output port.

A **CoupledProcessor** consists of two connected processors. The **Root** coupled model is composed of a generator, a coupled processor, a processor, and a collector. A *Job* event template is defined that defines the structure of events passed between DEVS instances, and the end condition is *false*, meaning that the simulation will run forever.

ALGORITHM 4: The Parallel DEVS simulator's "main loop".

```
Input: Model to simulate ($model$)
$flatten\_model(model)$;
$time \leftarrow 0$;
$state \leftarrow initialize\_state(model)$;
while not end_condition(time, state) do
    $imminents \leftarrow compute\_imminents(state)$;
    $selected\_component \leftarrow select\_imminent(imminents)$;
    $events \leftarrow compute\_output(selected\_component)$;
    $route\_events(events)$;
    $ext \leftarrow compute\_externals(state, events)$;
    for component in imminents $\cup$ ext do
        $compute\_next\_state(component, state)$;
    end
    $time \leftarrow time\_next(state)$;
end
```

To simulate Parallel DEVS models, the operational semantics of Parallel DEVS are described in [38], in the form of an abstract simulator. An efficient version of the simulation algorithm was introduced in [143] and implemented in tools such as *PythonPDEVS* [205] and *adevs* [148]. It is based on direct connection and flattening of the model hierarchy. The pseudocode for this algorithm is shown in Algorithm 4. A simulator computes the next state of the system (a "step") until its end condition is satisfied. Each step consists of the following phases:

1. Compute the set of atomic DEVS models whose internal transitions are scheduled to fire (imminent components).
2. Execute the output function for each imminent component, causing events to be put on their output ports.
3. Route events from output ports to input ports, translating them in the process by executing the transfer functions.
4. For each atomic DEVS model, determine the type of transition to execute, depending on it being imminent, receiving input, or both.
5. Execute, in parallel, all enabled internal, external, and confluent transition functions.
6. Compute, for each atomic DEVS model, the time of its next internal transition (specified by its time advance function).

During simulation, the value of an internal clock (the simulated time) is updated according to the information encoded in the transition functions.

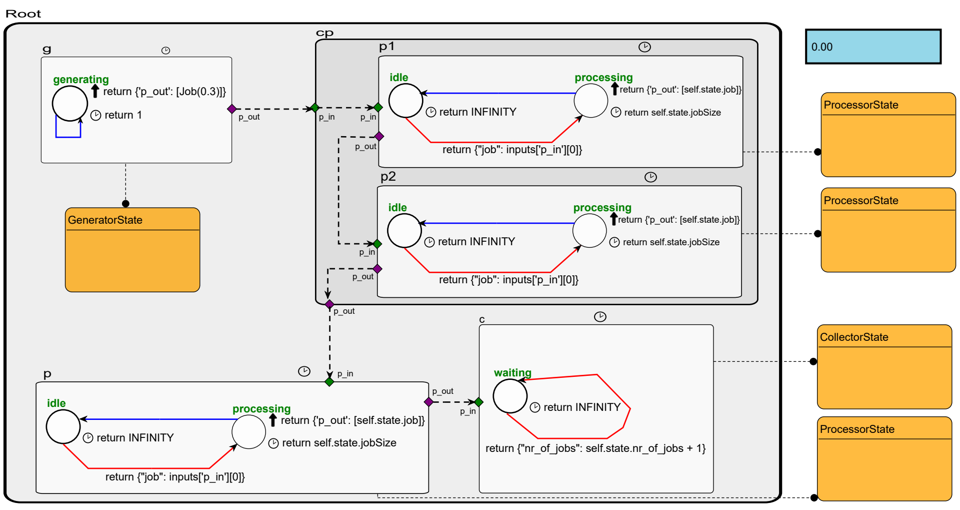

Root
g
generating
return {'p_out': [Job(0.3)]}
return 1
GeneratorState
cp
p1
idle
processing
return {'p_out': [self.state.job]}
return INFINITY
return self.state.jobSize
return {"job": inputs['p_in'][0]}
p2
p
c
waiting
return INFINITY
return {"nr_of_jobs": self.state.nr_of_jobs + 1}
p_out
p_in
0.00
ProcessorState
CollectorState

Figure 5.11: The runtime model of the producer-consumer DEVS model.

To represent Parallel DEVS model at runtime (*i.e.*, during simulation), a runtime language was created that extends the design language. This is necessary for a variety of reasons:

- The need to keep track of runtime information, such as:
  - The simulated time.
  - The state information for each atomic DEVS model, consisting of its attribute values and the name of the currently active state.
- To instantiate the references to coupled DEVS models, causing these references to be expanded, making the complete structure explicit.

The design metamodel is augmented with the necessary runtime information. An automatic exogenous transformation transforms any valid design of a DEVS model into a runtime model by retyping the elements and expanding all references to atomic DEVS and coupled DEVS models found in the root model: they are replaced by their definition. Figure 5.11 shows the resulting runtime model of the produce-consume example, of which the design is shown in Figure 5.10.

5.3.2 Debugging Operations

For Parallel DEVS, we define the following debugging operations:

- **[As-Fast-as-Possible Simulation]** In this mode, the simulation runs as fast as the underlying hardware can manage and the operating system will allow, until the end condition is satisfied. It is comparable to running program code, which is always run as fast as possible. At the end, the user can inspect the generated trace and any metrics collected.

- **[Real-Time Simulation]** In this mode, simulated time is synchronized with the wall-clock time. For debugging purposes, a scale factor can be applied to speed up or slow down simulation, while retaining the *linear* relation between simulated time and wall-clock time. A scale factor of 1 corresponds to real-time, while a scale factor smaller or greater than 1 slows down or speeds up simulation proportionally. State is observed throughout real-time simulation after each iteration of the simulation loop.

- **[Pause]** Pausing a simulation allows to inspect the current state of the system and enables other debugging operations, such as stepping and state modifications.

- **[Big Step and Small Step]** To transpose the notion of stepping, we turn to the simulation algorithm shown in Algorithm 4. We define two types of steps: a "big step" executes one simulation step (corresponding to an iteration of the *while*-loop). Inside of the *while*-loop, a number of simulation phases, or "small steps", are executed. Providing control and feedback at this level allows the user to gain more insight into the detailed model semantics.

- **[God Event]** A "god event" allows a user to change the value of a state variable of a particular atomic Parallel DEVS model. This operation changes the current state $(s_{i,curr}, e_{i,curr}) \in Q_i$ to a new state $(s_{i,new}, 0) \in Q_i$ where Q_i is the total state set of the atomic Parallel DEVS model i.

- **[Event Injection]** A user can schedule the injection of an event $(x, p) \in X_i$ at a specified (future) simulated time instant t. Once time t is reached, the atomic Parallel DEVS model i receives the input event x on its port p, triggering its external transition function.

- **[Breakpoints]** A user can specify a breakpoint in the form of a condition on the execution state of the Parallel DEVS simulation (including simulated time and current (total) state information). A breakpoint is a function that returns *True* when the simulation should pause, *False* in all other cases. When a breakpoint triggers, the user is notified of the name of the triggered breakpoint, and the current state of the simulation.

5.3.3 De- and Reconstructed Simulator

Figure 5.12 presents the Statecharts model of the debugging-enhanced Parallel DEVS simulator. Our goal was to support the operations presented in the previous subsection, while retaining the original execution semantics implemented by the simulator. This means we had to introduce points at which the algorithm can be interrupted (through stepping, breakpoints, etc.), as well as provide information on the state to the user of the debugger. Important with this formalism was to support the different "phases" (small steps) that occur throughout one "big step" (an iteration of the *while*-loop of the simulator). The Statecharts model consists of two main orthogonal components: `simulation_state` and `simulation_flow`.

The `simulation_state` determines in what execution mode the simulator is. It describes how the user can influence the control flow of the simulation. Four modes are defined: `paused`, `continuous`, `realtime`, and `big_step`. The simulator starts in

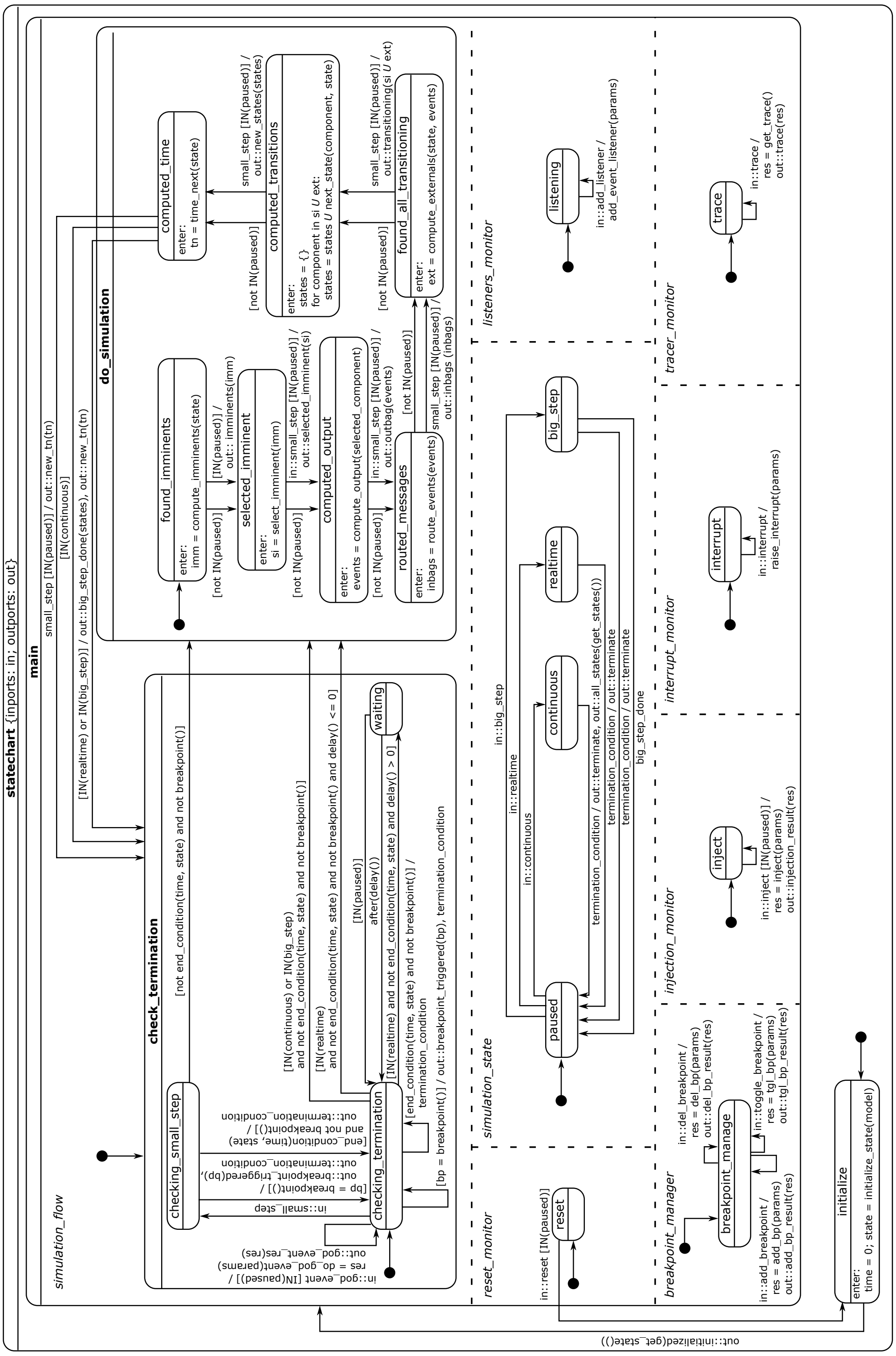

statechart {inports: in; outports: out}
main
simulation_flow
check_termination
checking_small_step
checking_termination
waiting
do_simulation
found_imminents
selected_imminent
computed_output
routed_messages
found_all_transitioning
computed_transitions
computed_time
reset_monitor
reset
simulation_state
paused
continuous
realtime
big_step
listeners_monitor
listening
breakpoint_manager
breakpoint_manage
injection_monitor
inject
interrupt_monitor
interrupt
tracer_monitor
trace
initialize
enter: time = 0; state = initialize_state(model)
out::initialized(get_state())
in::reset [IN(paused)]
in::continuous
in::realtime
in::big_step
in::add_listener / add_event_listener(params)
in::trace / res = get_trace() out::trace(res)
in::interrupt / raise_interrupt(params)
in::inject [IN(paused)] / res = inject(params) out::injection_result(res)
in::add_breakpoint / res = add_bp(params) out::add_bp_result(res)
in::del_breakpoint / res = del_bp(params) out::del_bp_result(res)
in::toggle_breakpoint / res = tgl_bp(params) out::tgl_bp_result(res)
termination_condition / out::terminate, out::all_states(get_states())
termination_condition / out::terminate
big_step_done
enter: imm = compute_imminents(state)
enter: si = select_imminent(imm)
enter: events = compute_output(selected_component)
enter: inbags = route_events(events)
enter: ext = compute_externals(state, events)
enter: states = {} for component in si U ext: states = states U next_state(component, state)
enter: tn = time_next(state)
small_step [IN(paused)] / out::new_tn(tn)
[IN(continuous)]
[IN(realtime) or IN(big_step)] / out::big_step_done(states), out::new_tn(tn)
[not IN(paused)]
[IN(paused)] / out:: imminents(imm)
in::small_step [IN(paused)] / out::selected_imminent(si)
in::small_step [IN(paused)] / out::outbag(events)
small_step [IN(paused)] / out::inbags (inbags)
small_step [IN(paused)] / out::transitioning(si U ext)
small_step [IN(paused)] / out::new_states(states)
[not end_condition(time, state) and not breakpoint()]
[IN(continuous) or IN(big_step) and not end_condition(time, state) and not breakpoint()]
[IN(realtime) and not end_condition(time, state) and not breakpoint() and delay() <= 0]
[IN(paused)]
after(delay())
[IN(realtime) and not end_condition(time, state) and delay() > 0]
[end_condition(time, state) and not breakpoint()] / termination_condition
[bp = breakpoint()] / out::breakpoint_triggered(bp), termination_condition
in::small_step
[bp = breakpoint()] / out::breakpoint_triggered(bp), out::termination_condition
[end_condition(time, state) and not breakpoint()] / out::termination_condition
in::god_event [IN(paused)] / res = do_god_event(params) out::god_event_res(res)

Figure 5.12: The de- and reconstructed Parallel DEVS simulator.

the `paused` state, and can only transition to one of the other states when a user event is received. Pausing is only possible when running the simulation in continuous mode or in realtime mode. The debugging-enhanced simulator does not allow pausing the simulation when computing a big step or small step, as that could leave the simulator in a (globally) inconsistent state. Rather, the simulator pauses automatically after every big step when in big step mode, and small steps are manually controlled (discussed below). When the simulation terminates (because the termination condition is satisfied, or due to a breakpoint), the simulation state transitions to the `paused` state from any of the other states. Detecting termination is encoded in the `simulation_flow` orthogonal component, discussed below. Based on the simulation state, the behaviour of the simulation flow changes.

The `simulation_flow` orthogonal component determines the flow of the simulation steps, and explicitly encodes the invocation order of the non-modal functions. This is an explicit representation of the control flow that was implemented in the *while*-loop in the coded implementation. The orthogonal component consists of two main components: `checking_termination` and `do_simulation`. In the `checking_termination` state, the simulator checks whether or not simulation should continue. It has self-transitions to manage god events (manual state changes), breakpoint detection, and checking whether the end condition was satisfied. By defining these events to only be processed (using transitions) in the `checking_termination` state, we ensure god events are only possible when a simulation "big step" is currently not executing. Breakpoints are detected here because they need to immediately pause the simulation. If the detection were performed in a different component, the detection result would have to be communicated to the `simulation_flow` component, which might react late and already be executing the next big step. To check whether the next simulation step can be executed, the `checking_termination` inspects the state of the orthogonal component `simulation_state`, and, if realtime simulation is selected, also checks whether the next simulation step can already be performed, or if the simulation algorithm needs to wait. In case we should wait for the next simulation step, the `waiting` state is entered. Otherwise, the `do_simulation` state is entered. If the simulation is executing in real time and the delay has not passed, the simulator has to wait. The control transitions to the `waiting` state, and a transition is scheduled to fire after the delay has passed. But, pausing needs to be possible while waiting. So, a transition is added which lets the control go back to the `check_termination` state when the user requests a pause. In any other cases, the next simulation step can be executed. The `do_simulation` composite state traverses through seven stages (encoded as states), each responsible for a part of the simulation algorithm. Linking back to the original code implementation, the `check_termination` state is the condition of the *while*-loop, whereas `do_simulation` is the body of the loop. If the simulation is not paused, the phases are executed without any communication to the user. When the simulation is paused, however, no automatic transition takes place. The user can then manually step through the simulation phases using the "small step" operation. After every small step, relevant information is sent to the user to be displayed in a debugging interface. The first small step is requested when the `simulation_flow` is in `checking_termination`. In that case, there are two possibilities: either the simulation is paused, or it has terminated. To detect these cases, a `checking_small_step` state was created, to which the system transitions when the first small step is requested. Only if the end condition is not satisfied and no breakpoints trigger is the first phase executed. At the end of the `do_simulation` state, there are three possibilities, each with a different

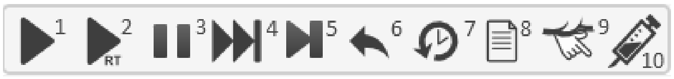


Figure 5.13: The debugging toolbar for Parallel DEVS.

response to the user. If simulation is running in realtime or a big step was being executed, all information is passed on to the user. Should simulation be done using a small step, only a part of the information needs to be sent, as all other information has already been sent during the previous steps. If simulation is running as fast as possible, no information is sent to the user, as it would only slow down simulation.

Six additional orthogonal components are modelled:

- The `reset_monitor` listens for user requests to reset the simulation, and allows it when the simulation is paused.
- The `breakpoint_manager` can add, toggle, and delete breakpoints, as well as list all currently defined breakpoints.
- The `injection_monitor` listens for user requests to inject an event on a port in the model, and allows such injections if the simulation is paused.
- The `interrupt_monitor` is used in real-time simulations for real-time interrupts from the environment.
- The `listeners_monitor` is used in real-time simulations for listening to events on an output port.
- The `tracer_monitor` can print the full textual trace of the simulation.

From this model, the code for the debugging-enhanced simulator can be generated using an appropriate Statecharts compiler.

Summary We demonstrated in this subsection that the debugging operations described in the previous subsection have been successfully added to the Parallel DEVS simulator. In the next subsection, we couple the instrumented simulator to a debugging environment.

5.3.4 Debugging Environment

To support debugging, a toolbar was added to the runtime environment, shown in Figure 5.13. By clicking on a button in this toolbar, a request is sent to the simulator, representing a particular command. The simulator will perform the requested action, if allowed. As a result, a reply is sent back to the debugging environment, which interprets the result and modifies the runtime model accordingly. The toolbar supports ten operations, explained in the following subsections.

Simulate

Simulates the model as-fast-as-possible, until either the end condition evaluates to true, one of the breakpoints triggers, or the user manually pauses the simulation. During as-fast-as-possible simulation, state changes are not visualized, as this would significantly slow down the simulation due to the overhead of visualization requests from the simulator to AToMPM. Moreover, visually keeping track of state changes in as-fast-as-possible simulation is difficult, and most likely not what this option is used for. It should rather be used to quickly reach a breakpoint or the end of simulation.

Realtime Simulate

Simulates the model in real-time. This means that the scheduler will try to meet all real-time deadlines, as specified in the time advance functions of the states in the atomic DEVS models. The values returned by these functions are interpreted as seconds. It is possible to pass a *scale factor* to real-time simulation. This floating point number specifies how much faster (if the value is larger than 1) or slower (if the value is smaller than 1) the simulation should be run. In other words, the scale factor specifies the linear relationship between the wall clock time and the simulated time: if it is 1, they are equal, if it is 2, simulated time advances twice as fast as wall clock time. During real-time simulation, the model's visualization is updated, meaning that the user can track the simulation process.

Pause

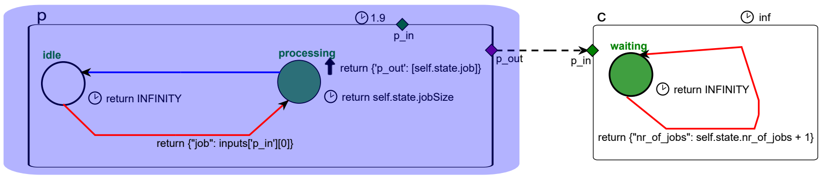
Pressing the pause button will result in a running (as-fast-as-possible or real-time) simulation being paused as soon as possible. In as-fast-as-possible simulation, simulation is stopped after the currently executing big step is finished. In real-time simulation, it is additionally possible that the simulator is waiting until its next transition should be fired. The simulation is then paused in this waiting phase. When the simulation is paused, the current state of the model is visualized. Resuming is done by either stepping, as-fast-as-possible simulation, or realtime simulation. Additionally, the scale factor can be changed when starting a realtime simulation.

Big Step

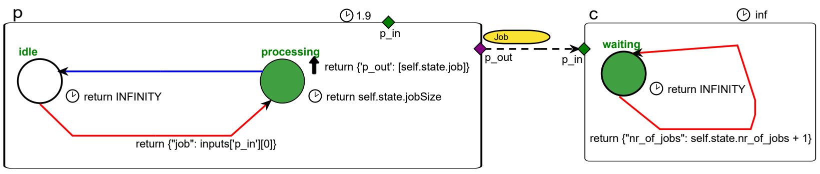
When the simulation is paused, the user can choose to continue the simulation by *stepping*. A big step computes output functions, the internal, external, and confluent transitions, and schedules the next internal transition function. The simulation state at the end of the big step is visualized in the simulation environment.

Small Step

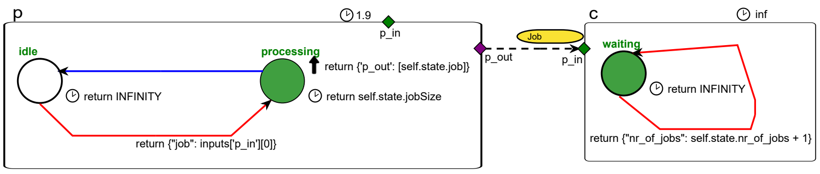
A big step consists of six distinct phases. A small step allows to visualize these phases, called *small steps*. We list below for each phase what happens, and how it is visualized. The phases are shown in Figure 5.14.


p
1.9
p_in
idle
processing
return {'p_out': [self.state.job]}
return INFINITY
return self.state.jobSize
return {"job": inputs['p_in'][0]}
p_out
c
inf
p_in
waiting
return INFINITY
return {"nr_of_jobs": self.state.nr_of_jobs + 1}

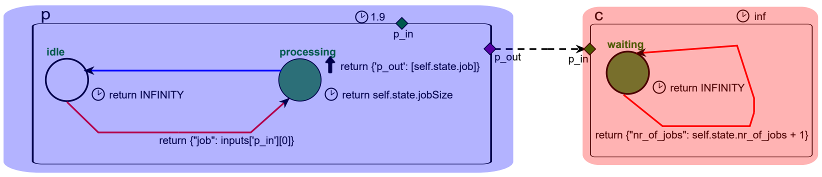
(a) Finding imminent components.


p
1.9
p_in
Job
idle
processing
return {'p_out': [self.state.job]}
return INFINITY
return self.state.jobSize
return {"job": inputs['p_in'][0]}
p_out
c
inf
p_in
waiting
return INFINITY
return {"nr_of_jobs": self.state.nr_of_jobs + 1}

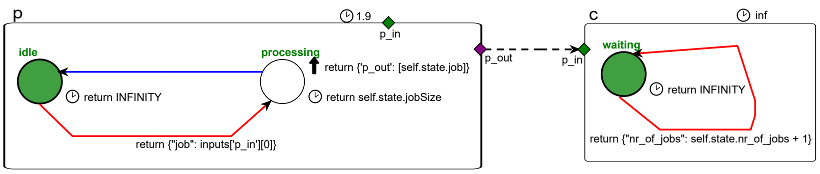
(b) Generate their output.


p
1.9
p_in
Job
idle
processing
return {'p_out': [self.state.job]}
return INFINITY
return self.state.jobSize
return {"job": inputs['p_in'][0]}
p_out
c
inf
p_in
waiting
return INFINITY
return {"nr_of_jobs": self.state.nr_of_jobs + 1}

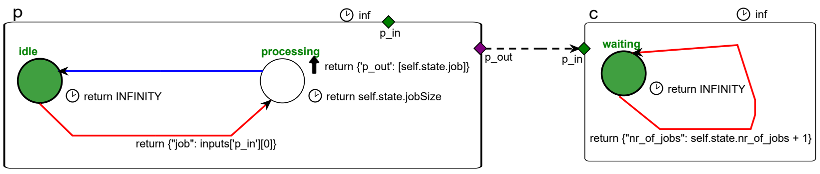
(c) Output routed from output to input port.


p
1.9
p_in
idle
processing
return {'p_out': [self.state.job]}
return INFINITY
return self.state.jobSize
return {"job": inputs['p_in'][0]}
p_out
c
inf
p_in
waiting
return INFINITY
return {"nr_of_jobs": self.state.nr_of_jobs + 1}

(d) Mark transition function to execute.


p
1.9
p_in
idle
processing
return {'p_out': [self.state.job]}
return INFINITY
return self.state.jobSize
return {"job": inputs['p_in'][0]}
p_out
c
inf
p_in
waiting
return INFINITY
return {"nr_of_jobs": self.state.nr_of_jobs + 1}

(e) Execute applicable transition function.


p
inf
p_in
idle
processing
return {'p_out': [self.state.job]}
return INFINITY
return self.state.jobSize
return {"job": inputs['p_in'][0]}
p_out
c
inf
p_in
waiting
return INFINITY
return {"nr_of_jobs": self.state.nr_of_jobs + 1}

(f) Set time of next internal transition.

Figure 5.14: Sequence of small steps, forming a single big step.

1. Computing all imminent components (those DEVS models whose internal transitions are scheduled to execute). The imminent components are highlighted in blue.
2. Executing output functions for each imminent component, which results in events being put on their output ports. To visualize this, an event is instantiated on the position of the output port.
3. Routing events from output ports to input ports, while executing the transfer function. Visually, the event instance is moved to the position of the input port.
4. Deciding which models will execute their external (or, in the case it is an imminent component, confluent) transition function as a result of events on their input ports. Atomic DEVS models that will execute their external transition function are coloured red, those that will execute their confluent transition function are coloured purple.
5. Executing, in parallel, all enabled internal, external, and confluent transition functions. The resulting state is highlighted in green, and the new values of its instance variables are displayed.
6. Computing, for each atomic DEVS model, the time at which its internal transition function is scheduled (which is specified in its *time advance* function). This value is displayed next to a clock icon on top of each atomic DEVS model.

Reset

Resets the model and the simulation to their initial state. The initial state is visualized.

Show Trace

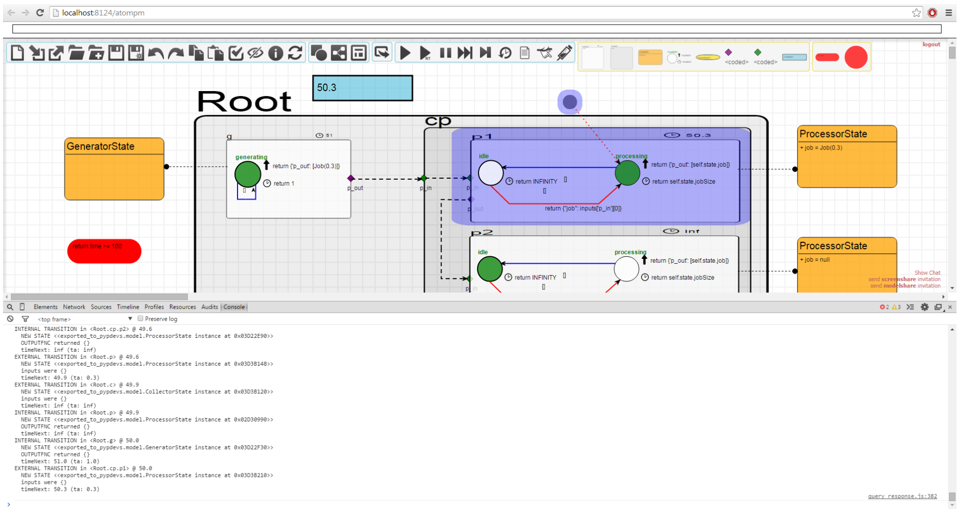
During simulation, a textual trace is generated. By clicking this button, the trace will be displayed in the debugging environment.

Insert God Event

A "god event" allows to manually change the value of a state variable. With this functionality, the environment allows to (visually) inspect and modify the internal state of DEVS models during a debugging session.

Inject Events

Events can be injected when the simulation is paused by instantiating an event and connecting it to the port at which the event should be injected. Optionally, the user can specify the simulation time at which the event should be injected. By default, this is "now" (*i.e.*, the current simulation time). Events are not injected until this button is clicked. Once simulation is resumed, injected events are removed from the visual model. Once the time for their injection is reached, they will reappear on the appropriate port.



```
INTERNAL TRANSITION in <Root.cp.p2> @ 49.6
  NEW STATE <<exported_to_pypdevs.model.ProcessorState instance at 0x03D22E90>>
  OUTPUTFNC returned {}
  timeNext: inf (ta: inf)
EXTERNAL TRANSITION in <Root.p> @ 49.6
  NEW STATE <<exported_to_pypdevs.model.ProcessorState instance at 0x03D3B148>>
  inputs were {}
  timeNext: 49.9 (ta: 0.3)
EXTERNAL TRANSITION in <Root.c> @ 49.9
  NEW STATE <<exported_to_pypdevs.model.CollectorState instance at 0x03D3B120>>
  inputs were {}
  timeNext: inf (ta: inf)
INTERNAL TRANSITION in <Root.p> @ 49.9
  NEW STATE <<exported_to_pypdevs.model.ProcessorState instance at 0x02D30990>>
  OUTPUTFNC returned {}
  timeNext: inf (ta: inf)
INTERNAL TRANSITION in <Root.g> @ 50.0
  NEW STATE <<exported_to_pypdevs.model.GeneratorState instance at 0x03D22F30>>
  OUTPUTFNC returned {}
  timeNext: 51.0 (ta: 1.0)
EXTERNAL TRANSITION in <Root.cp.p1> @ 50.0
  NEW STATE <<exported_to_pypdevs.model.ProcessorState instance at 0x03D3B210>>
  inputs were {}
  timeNext: 50.3 (ta: 0.3)
```

Figure 5.15: The debugging environment for Parallel DEVS.

Breakpoints

We offer two ways of defining a breakpoint, which pauses the simulation automatically when the specified condition is fulfilled. A *global breakpoint* models such a condition. The simulation breaks whenever the condition is satisfied. A *local breakpoint* also models a state condition, but is additionally connected to a state. The breakpoint will only trigger when both the condition is satisfied and the simulation enters the state the breakpoint is connected to. This is comparable to breakpoints in code debugging that are associated to a specific line of code. The user can configure whether the breakpoint should be automatically disabled after triggering.

The debugging environment is shown in Figure 5.15. A breakpoint was triggered, represented by the black circle highlighted in blue. One of the components is about to execute its internal transition function (the result of a small step). The model is flattened to show its complete structure and the current state is visualized by highlighting, for each component, the discrete state the component is in. The full textual state trace up to that point in simulation is shown in the console.

Summary

This section demonstrates how debugging support can be implemented for Parallel DEVS. The language exhibits the following semantic features of the feature diagram presented in Figure 4.2:

- It implements **discrete-event** semantics.
- It is **deterministic**.
- It has **static structure**.

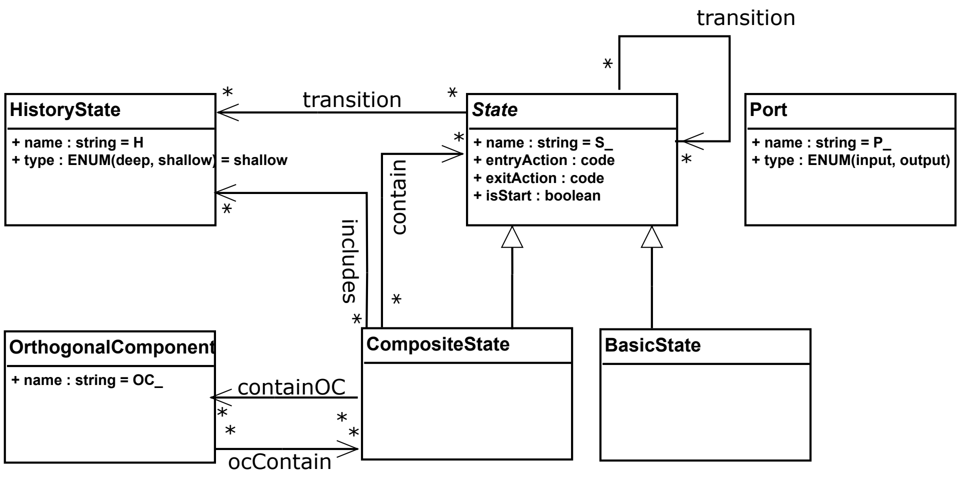


Figure 5.16: The metamodel for the Statecharts language.

The implementation of discrete-event semantics, as well as its decomposition in nested blocks, are the most prominent features of this formalism. In this section, we have demonstrated that our approach can support debugging for languages exhibiting these features.

5.4 Statecharts

Statecharts is used throughout this thesis to model the timed, reactive, autonomous behaviour of (debugging-enhanced) simulation kernels. It is a causal, timed (discrete-event), static-structure, deterministic formalism.

5.4.1 Syntax and Semantics

The metamodel for Statecharts is shown in Figure 5.16. A Statecharts model consists of the set of the system's discrete states, and transitions between those states that model the dynamics of the system.

A state has the following attributes:

- A unique *name* within its enclosing scope.
- An optional *entry action* that is executed when the state is entered.
- An optional *exit action* that is executed when the state is exited.
- A boolean value *isStart* to denote whether the state is the default state within its enclosing scope.

States can be composed hierarchically in *composite states* as well as orthogonally in *parallel regions*. Within each composite state, exactly one state is the default state. When the composite state is entered, its default state is entered as well (except when a history state is present in the composite state, see below). A parallel region is composed of multiple composite states: upon entering the orthogonal region, all its child states are entered as well. *History states* are used to remember the active child state of a composite state. When the composite state is re-entered, the state that was originally active is restored. History states can be *shallow*, remembering only the direct child of its parent state that was active, or they can be *deep*, in which case they remember all active descendant states.

Transitions between states model the dynamics of the system. A transition is *triggered* by an external event (coming from the environment), an internal event that was raised by an orthogonal component, a timeout, or it is spontaneous. Events dictate the passing of time in Statecharts: the clock is synchronized either with the timeout event, or with the time at which an external interrupt arrives. An optional *guard* specifies an additional runtime condition that must be satisfied in order for the transition to be enabled. An optional *action* is executed when the transition is executed. When a transition is executed, its source state (and its descendants) is (are) exited, and its target state (and its descendants) is (are) entered. An algorithm determines the least common ancestor of the source and target states, and also exits any states up to (but excluding) the least common ancestor, and enters any states down from the least common ancestor to the target state.

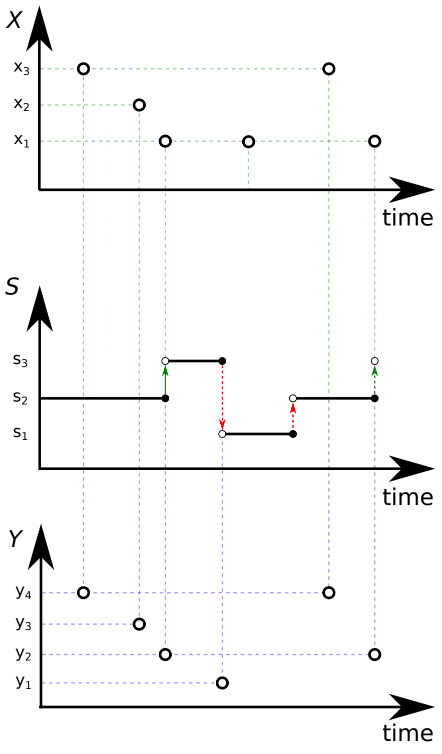
As part of a transition's execution, or when entering or exiting a state, an action can be executed. An action is specified in an action language with imperative constructs for modifying runtime variables (this language can be similar to the action language for which a debugger was presented before, refer back to Section 5.1). Actions can also *raise* an event. The *scope* of the raised event is either local, in which case it can trigger transitions in components orthogonal to the component in which the event was raised, or the event can be raised on an output port, in which it can be "sensed" by the environment. The interface with the environment is specified using a set of named *Ports*. A port can either be used to receive events from the environment (input ports) or send events to the environment (output ports).

The behaviour of Statecharts models is summarized in Figure 5.17. The input-output segments are similar to those of DEVS: there is a set of possible inputs *X*, a set of possible outputs Y and a state set *S*. The state of the system evolves over time according to its autonomous behaviour and the events it receives from the environment. One possible state evolution is plotted: some state changes occur when an event is received from the environment, others occur autonomously (after a timeout). Output to the environment is optionally generated. A possible model, capturing all possible state evolutions of such a system, is shown in Figure 5.17b. The model has one input port in and one output port out. Three top-level states are modelled: s_1, s_2 (the default state), and s_3. s_2 has three substates W (the default state), U, and D. We distinguish three different types of trigger for transitions: an external event (triggering, for example, the transition from W to U), an internal event (triggering, for example, the transition from s_3 to s_2, in this case a timeout) and a spontaneous transition with a condition (triggering, for example, the transition from $s3$ to $s1$). Actions can be modelled, but are optional.

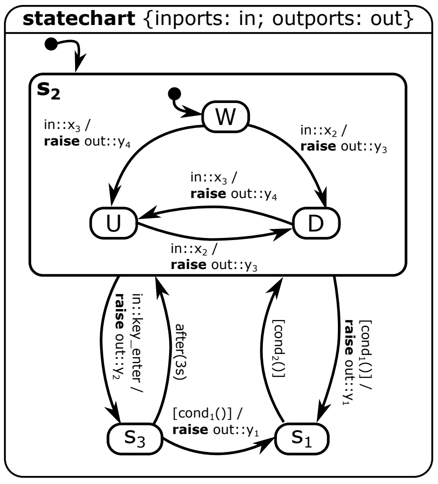
The simulation algorithm for Statecharts is presented in Algorithm 5. Statecharts is a "Big-Step Modelling Language" (BSML) [51]. Its semantics are controlled by a series of *big steps*, which consist of smaller computation steps. The execution of a BSML model is a sequence of big steps. Before a big step starts, and after a big step has executed, it

ALGORITHM 5: The Statecharts simulator's "main loop".

```
Input: Model to simulate ($model$)
$time \leftarrow 0$;
$state \leftarrow initialize\_state(model)$;
while not end_condition(time, state) do
    $\delta_t \leftarrow time\_next(model, state)$;
    if $\delta_t = \infty$ then
        $wait\_for\_input()$;
        $break$;
    end
    $decrease\_event\_time(state, delta\_t)$;
    $due \leftarrow get\_due\_events(state)$;
    for event in due do
        $next\_big\_step(event)$;
        $reset\_combo\_step()$;
        $reset\_small\_step()$;
        while True do
            $next\_combo\_step()$;
            $candidates \leftarrow get\_candidates(state)$;
            if candidates then
                $next\_small\_step()$;
                $candidate \leftarrow select\_candidate(candidates)$;
                $execute\_transition(candidate)$;
                $set\_small\_stepped()$;
                $set\_combo\_stepped()$;
            end
            if $has\_combo\_stepped()$ then
                $set\_big\_stepped()$;
            else
                $output\_events(state)$;
                $add\_internal\_events(state)$;
                $break$;
            end
        end
    end
    $time \leftarrow time + \delta_t$;
end
```



(a) Input-output segments of a discrete-event system modelled with Statecharts.


statechart {inports: in; outports: out}
s2
W
U
D
in::x3 / raise out::y4
in::x2 / raise out::y3
in::x3 / raise out::y4
in::x2 / raise out::y3
in::key_enter / raise out::y2
after(3s)
[cond2()]
[cond1()] / raise out::y1
[cond1()] / raise out::y1
s3
s1

(b) Example Statecharts model, a possible model of the behaviour described by the input-output segments on the left.

Figure 5.17: The example model and its input-output segments trace.

can sense input from the environment, but during a big step, no new input to the model can be provided. In Statecharts, a big step begins when an external event is received or a timeout occurs. A big step consists of zero or more *small steps*. A small step consists of exactly one execution of a transition. Small steps are grouped in *combo steps*: a combo step executes small steps until no transition is enabled by the current event from the environment or locally raised events in previous small steps. Combo steps are useful if events raised by concurrent transitions (in small steps) can be visible throughout the combo step, or to execute multiple spontaneous transitions.

The algorithm is implemented using the following functions:

- *initialize_state()* initializes the time and the state of the simulation.
- *end_condition()* returns *True* if the simulation's end condition is satisfied. For Statecharts models, which often describe systems that run indefinitely, this function often returns *False*.
- *time_next()* returns the time at which the next event is scheduled: in case an external event was received, that time is *now*, otherwise, it is the time at which the next timeout occurs.

- *wait_for_input()* blocks until input from the environment is received.
- *decrease_event_time()* decreases the "remaining time" of all running timers.
- *get_due_events()* returns the due events, which are either internally raised events, events received from the environment, or timeouts.
- *get_candidates()* returns the transitions that are triggered by the due event.
- *select_candidate()* selects one transition from the set.
- *next_big_step()* starts a new big step.
- *next_combo_step()* starts a new combo step, which receives the events that were raised in the previous combo step, and can trigger transitions in this combo step.
- *next_small_step()* starts a new small step.
- *reset_combo_step()* resets the combo step state, clearing any internally events.
- *reset_small_step()* resets the small step state, clearing any internally raised events.
- *set_small_stepped()* sets the *stepped* flag for the currently executing small step to *True*.
- *set_combo_stepped()* sets the *stepped* flag for the currently executing combo step to *True*.
- *set_big_stepped()* sets the *stepped* flag for the currently executing big step to *True*.
- *has_combo_stepped()* returns the value of the *stepped* flag for the currently executing combo step.

The runtime metamodel for the Statecharts language extends the design metamodel's *State* class with an *isCurrent* attribute, which signifies the state is active. An active state is visualized by highlighting it in green.

5.4.2 Debugging Operations

We define the following debugging operations for debugging Statecharts models:

- **[As-Fast-as-Possible Simulation]** In this mode, the simulation runs as fast as the underlying hardware can manage and the operating system will allow, until the end condition is satisfied. It is comparable to running program code, which is always run as fast as possible. At the end, the user can inspect the runtime state in which the system ended.
- **[Real-Time Simulation]** In this mode, simulated time is synchronized with the wall-clock time. For debugging purposes, a scale factor can be applied to speed up or slow down simulation, while retaining the *linear* relation between simulated time and wall-clock time. A scale factor of 1 corresponds to real-time, while a scale factor smaller or greater than 1 slows down or speeds up simulation proportionally. State is observed throughout real-time simulation after each iteration of the simulation loop.

- **[Pause]** Pausing a simulation allows to inspect the current state of the system and enables other debugging operations, such as stepping and state modifications.
- **[Time Step, Big Step, Combo Step, and Small Step]** To transpose the notion of stepping, we turn to the simulation algorithm shown in Algorithm 5. In the case of Statecharts, we implement four types of steps:
  - A *time step* executes one step of the system which changes its simulated time value and consists of a number of big steps.
  - A *big step* executes one step of the system for an external event (since events can arrive at the same time, multiple big steps can occur at the same time instant, forming a time step).
  - A *combo step* executes transitions until none are enabled.
  - A *small step* executes one enabled transition.
- **[God Event]** A "god event" allows a user to change the value of a state variable of the Statecharts model's memory.
- **[Event Injection]** A user can inject an event on one of the input ports of the Statecharts model at the "current" time.
- **[Breakpoints]** A user can specify a breakpoint in the form of a condition on the execution state of the Statecharts simulation (including simulated time and current (total) state information). In contrast to code debugging, we do not allow the simulation to break on a specific line of code. Instead, the simulation can be paused automatically when a condition on the simulation state is satisfied. We allow the user to scope the breakpoint, by only evaluating the breakpoint when a specified state is entered. A breakpoint is a function that returns *True* when the simulation should pause, *False* in all other cases.

5.4.3 De- and Reconstructed Simulator

Figure 5.18 presents the Statecharts model of the debugging-enhanced Statecharts simulator. Our goal was to support the operations presented in the previous subsection, while retaining the original execution semantics implemented by the simulator. This means we had to introduce points at which the algorithm can be interrupted (through stepping, breakpoints, etc.), as well as provide information on the state to the user of the debugger. Important in this formalism is the decomposition of instance models into hierarchical states and orthogonal regions, that exchange information through events. This requires the simulator to be broken up into multiple nested loops for ever-smaller "steps" that need to be supported. The Statecharts model consists of two main orthogonal components: `simulation_state` and `simulation_flow`.

The `simulation_state` determines in what execution mode the simulator is. It describes how the user can influence the control flow of the simulation. Two main modes are defined: `paused` and `running`. Within `running`, six substates are defined: `continuous`, `realtime`, `time_step`, `big_step`, `combo_step`, and `small_step`.

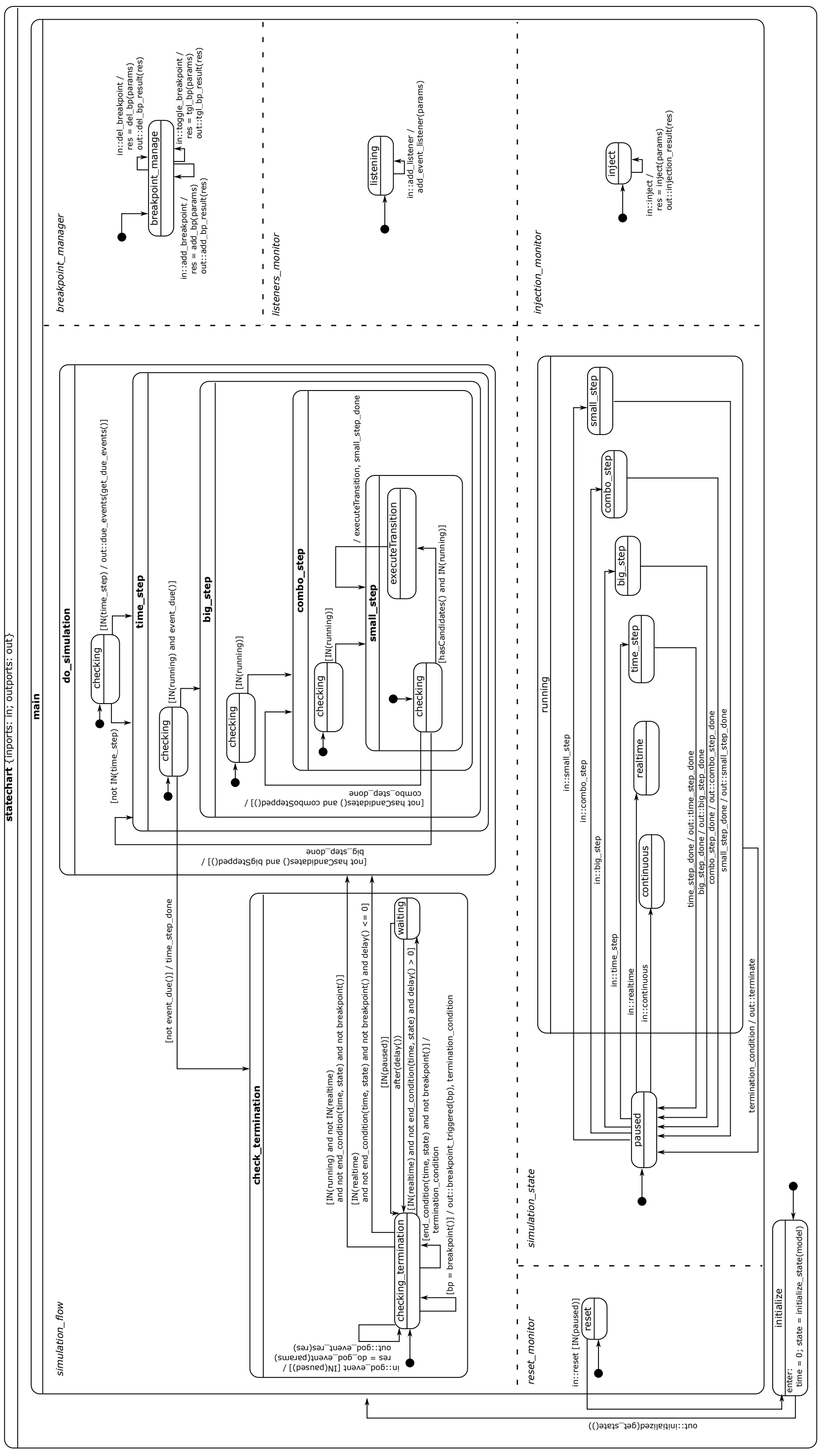

statechart {inports: in; outports: out}
main
breakpoint_manager
breakpoint_manage
in::add_breakpoint / res = add_bp(params) out::add_bp_result(res)
in::del_breakpoint / res = del_bp(params) out::del_bp_result(res)
in::toggle_breakpoint / res = tgl_bp(params) out::tgl_bp_result(res)
listeners_monitor
listening
in::add_listener / add_event_listener(params)
injection_monitor
inject
in::inject / res = inject(params) out::injection_result(res)
simulation_flow
do_simulation
checking
[IN(time_step)] / out::due_events(get_due_events())
[not IN(time_step)]
time_step
[IN(running) and event_due()]
big_step
[IN(running)]
combo_step
[IN(running)]
small_step
executeTransition
/ executeTransition, small_step_done
[hasCandidates() and IN(running)]
[not hasCandidates() and comboStepped()] / combo_step_done
[not hasCandidates() and bigStepped()] / big_step_done
[not event_due()] / time_step_done
check_termination
checking_termination
waiting
[IN(running) and not IN(realtime) and not end_condition(time, state) and not breakpoint()]
[IN(realtime) and not end_condition(time, state) and not breakpoint() and delay() <= 0]
[IN(paused)]
after(delay())
[IN(realtime) and not end_condition(time, state) and delay() > 0]
[end_condition(time, state) and not breakpoint()] / termination_condition
[bp = breakpoint()] / out::breakpoint_triggered(bp), termination_condition
in::god_event [IN(paused)] / res = do_god_event(params) out::god_event_res(res)
simulation_state
running
small_step
combo_step
big_step
time_step
realtime
continuous
paused
in::small_step
in::combo_step
in::big_step
in::time_step
in::realtime
in::continuous
time_step_done / out::time_step_done
big_step_done / out::big_step_done
combo_step_done / out::combo_step_done
small_step_done / out::small_step_done
termination_condition / out::terminate
reset_monitor
reset
in::reset [IN(paused)]
initialize
enter: time = 0; state = initialize_state(model)
out::initialized(get_state())

Figure 5.18: The de- and reconstructed Statecharts simulator.

The simulator starts in the `paused` state, and can only transition to one of the other states when a user event is received. Pausing is only possible when running the simulation in continuous mode or in realtime mode. The debugging-enhanced simulator does not allow pausing the simulation when computing a big step or small step, as that could leave the simulator in a (globally) inconsistent state. Rather, the simulator pauses automatically after every time step when in time step mode, after every big step when in big step mode, after every combo step when in combo step mode, and after every small step when in small step mode. When the simulation terminates (because the termination condition is satisfied, or due to a breakpoint), the simulation state transitions to the `paused` state from any of the other states. Detecting termination is encoded in the `simulation_flow` orthogonal component, discussed below. Based on the simulation state, the behaviour of the simulation flow will change.

The `simulation_flow` orthogonal component determines the flow of the simulation steps, and explicitly encodes the invocation order of the non-modal functions. This is an explicit representation of the control flow that was implemented in the *while*-loops in the coded implementation. The orthogonal component consists of two main components: `check_termination` and `do_simulation`. In the `check_termination` state, the simulator checks whether or not simulation should continue. It has self-transitions to manage god events (manual state changes), breakpoint detection, and checking whether the end condition was satisfied. By defining these events to only be processed (using transitions) in the `checking_termination` state, we ensure god events are only possible when a simulation "time step" is currently not executing. Breakpoints are detected here because they need to immediately pause the simulation. If the detection were performed in a different component, the detection result would have to be communicated to the `simulation_flow` component, which might react late and already be executing the next time step. To check whether the next simulation step can be executed, the `checking_termination` inspects the state of the orthogonal component `simulation_state`, and, if realtime simulation is selected, also checks whether the next simulation step can already be performed, or if the simulation algorithm needs to wait. In case we should wait for the next simulation step, the `waiting` state is entered. Otherwise, the `do_simulation` state is entered. If the simulation is executing in real time and the delay has not passed, the simulator has to wait. The control transitions to the `waiting` state, and a transition is scheduled to fire after the delay has passed. But, pausing needs to be possible while waiting. So, a transition is added which lets the control go back to the `check_termination` state when the user requests a pause. In any other cases, the next simulation step can be executed. The `do_simulation` composite state contains a hierarchy of states, one for each type of step. Control starts in the `checking` state, which checks whether a time step was requested. If it was, it sends all currently due events as debug information. Otherwise, it transitions normally to the `time_step` state. In the `time_step` state, the algorithm checks whether the simulation is still running and an event is due. If no event is due, it transitions back to the `check_termination` state, since the time step ended. If an event is due, the next big step is executed. The `big_step` state simply transitions to the `combo_step` state if the simulation is running. The `combo_step` state simply transitions to the `small_step` state if the simulation is running. The `small_step` state checks whether the simulation is running and there are candidate transformations left. If there are, it executes one of them and generates the *small_step_done* event. If there are no more candidates left, it transitions back to

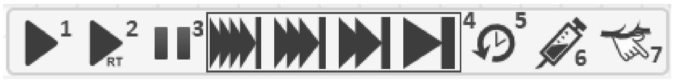


Figure 5.19: The debugging toolbar for Statecharts.

the combo_step state if a combo step was executed, and to the big_step state if a big step was executed. These states implement the nested *while*-loops of the original algorithm.

Four more orthogonal regions implement several debugging functions:

- The reset_monitor listens for user requests to reset the simulation, and allows it when the simulation is paused.
- The breakpoint_manager can add, toggle, and delete breakpoints, as well as list all currently defined breakpoints.
- The injection_monitor listens for user requests to inject an event on an input port of the model.
- The listeners_monitor is used in real-time simulations for listening to events on an output port.

From this model, the code for the debugging-enhanced simulator can be generated using an appropriate Statecharts compiler.

Summary We demonstrated in this subsection that the debugging operations described in the previous subsection have been successfully added to the Statecharts simulator. In the next subsection, we couple the instrumented simulator to a debugging environment.

5.4.4 Debugging Environment

To support debugging, a toolbar was added to the runtime environment, shown in Figure 5.19. By clicking on a button in this toolbar, a request is sent to the simulator, representing a particular command. The simulator will perform the requested action, if allowed. As a result, a reply is sent back to the debugging environment, which interprets the result and modifies the runtime model accordingly. The toolbar supports ten operations, explained in the following subsections.

Simulate

Simulates the model as-fast-as-possible, until either the end condition is satisfied, one of the breakpoints triggers, or the user manually pauses the simulation. During as-fast-as-possible simulation, state changes are not visualized, as this would significantly slow down the simulation due to the overhead of visualization requests from the simulator to AToMPM. Moreover, visually keeping track of state changes in as-fast-as-possible simulation is difficult, and most likely not what this option is used for. It should rather be used to quickly reach a breakpoint or the end of simulation.

Realtime Simulate

Simulates the model in real-time. This means that the scheduler will try to meet all real-time deadlines, as specified in the timed transitions of the Statecharts model. It is possible to pass a *scale factor* to real-time simulation. This floating point number specifies how much faster (if the value is larger than 1) or slower (if the value is smaller than 1) the simulation should be run. In other words, the scale factor specifies the linear relationship between the wall clock time and the simulated time: if it is 1, they are equal, if it is 2, simulated time advances twice as fast as wall clock time. During real-time simulation, the model's visualization is updated, meaning that the user can track the simulation process.

Pause

Pressing the pause button will result in a running (as-fast-as-possible or real-time) simulation being paused as soon as possible. In as-fast-as-possible simulation, simulation is stopped after the currently executing big step is finished. In real-time simulation, it is additionally possible that the simulator is waiting until its next transition should be fired. The simulation is then paused in this waiting phase. When the simulation is paused, the current state of the model is visualized. Resuming is done by either stepping, as-fast-as-possible simulation, or realtime simulation. Additionally, the scale factor can be changed when starting a realtime simulation.

Time Step, Big Step, Combo Step, Small Step

When the simulation is paused, the user can choose to continue the simulation by *stepping*. The time step, big step, combo step, and small step buttons execute one step of the specified type. When the step has completed, the new state of the system is visualized.

Reset

Resets the model and the simulation to their initial state. The initial state is visualized.

Inject Events

Events can be injected on an input port. The event has two parameters: a *name* and a list of parameter values that can be received by the transition triggered by the event.

Insert God Event

A "god event" allows to manually change the value of a state variable.

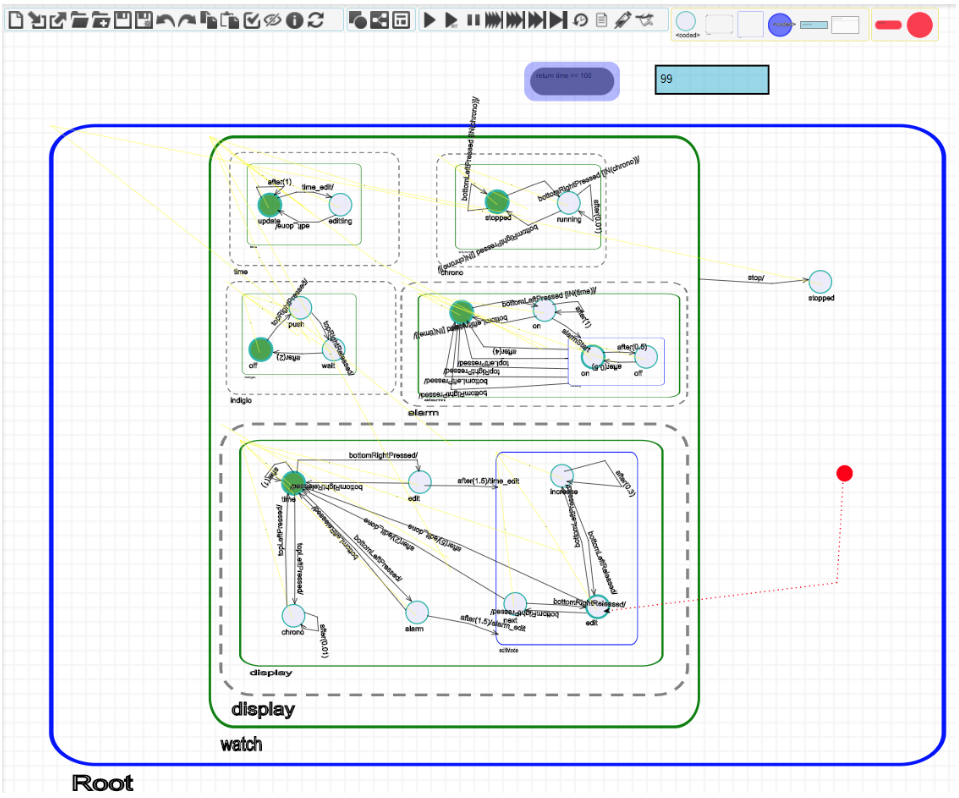


Figure 5.20: The debugging environment for the Statecharts formalism.

Breakpoints

We offer two ways of defining a breakpoint, which pauses the simulation automatically when the specified condition is fulfilled. A *global breakpoint* models such a condition. The simulation breaks whenever the condition is satisfied. A *local breakpoint* also models a state condition, but is additionally connected to a state. The breakpoint will only trigger when both the condition is satisfied and the simulation enters the state the breakpoint is connected to. This is comparable to breakpoints in code debugging that are associated to a specific line of code. The user can configure whether the breakpoint should be automatically disabled after triggering.

Figure 5.20 shows the debugging environment for Statecharts. It displays the runtime state of the model, in this case the behavioural model of a digital watch. The current state is visualized by highlighting the active state(s). The current simulated time is shown as well. A (global) breakpoint was triggered, and is highlighted in blue.

Debugging the Debugger Since we create a debugger for Statecharts, and its behaviour is defined as a Statecharts model, a particular application of this debugger would be to debug itself. We need three components for this to work, and place them in the architecture shown in Figure 4.13:

1. The debugging enhanced simulator for the Statecharts language, shown in Figure 5.18. This simulator is then instructed to simulate its own definition (a Statecharts model).

2. The visual debugging environment for Statecharts, with the visual representation (runtime model) of the model loaded.

3. The visual debugging environment, loaded a second time, but now acting as a model-specific visualization UI. In that environment, we can load a different Statecharts model to debug. The state of the debugger will be appropriately update in the *Debugging UI* component.

In this case, we have a nested simulation (and debugging session) of Statecharts models: the debugging UI (as model-specific visualization UI) loads a Statecharts model to be debugged, but itself is being debugged by a second debugging UI (and simulator) for the same formalism. This is not an issue, as long as proper translation functions are defined to "unpack" the events coming from the simulator. In the debugging UI, state updates can be displayed as they are received from the simulator, but in the model-specific visualization UI, the state information inside that event needs to be unpacked in order to visualize the state of the model that is being debugged there. Incidentally, this scales to arbitrary numbers of "debuggers inside of debuggers", if such use cases would exist.

Summary

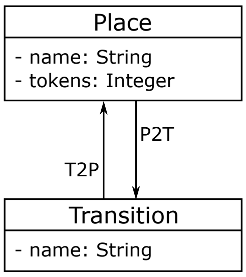
This section demonstrates how debugging support can be implemented for Statecharts. The language exhibits the following semantic features of the feature diagram presented in Figure 4.2:

- It implements **discrete-event** semantics.
- It allows for **environment interaction** (through input/output events).
- It is **deterministic**.
- It has **static structure**.

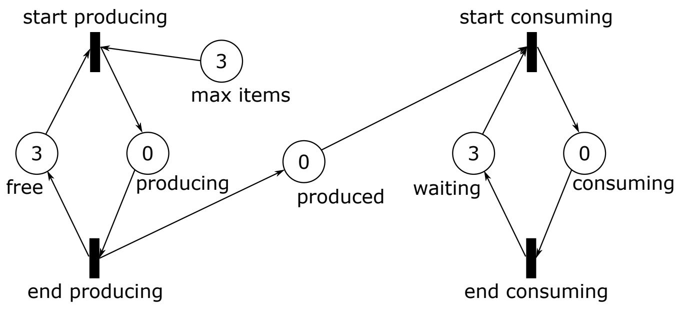
The implementation of discrete-event semantics, its decomposition in nested states, as well as its interaction with the environment, are the most prominent features of this formalism. In this section, we have demonstrated that our approach can support debugging for languages exhibiting these features.

5.5 Petrinets

Petrinets is a formalism that allows to express non-deterministic behaviour natively. From a debugging point of view, it is an interesting formalism to create a debugger for, since its execution semantics consider all possible execution paths of the model, not just one,



(a) The design metamodel for the Petrinets formalism.



(b) Example Petrinets model for a producer-consumer system.

Figure 5.21: The language definition and example model for the Petrinets language.

as is the case for the other formalisms presented in this chapter. It is a causal, untimed, static-structure, non-deterministic formalism.

5.5.1 Syntax and Semantics

The syntax of the design language of Petrinets is shown as a metamodel in Figure 5.21a. Petrinets models consist of:

- a number of named places, that can contain tokens (an integer value);
- a number of named transitions;
- a number of directed edges connecting places to transitions and transitions to places that have a weight (an integer value).

An example model is shown in Figure 5.21b. It models a simple producer-consumer system, where products are produced and consumed in a non-deterministic order.

ALGORITHM 6: The Petrinets analyser's "main loop".

```
Input: Model to simulate ($model$)
$state \leftarrow initialize\_state(model)$;
while not end_condition(model, state) do
    $unexplored\_markings \leftarrow find\_unexplored\_markings(model, state)$;
    $chosen\_marking \leftarrow choose\_marking(unexplored\_markings)$;
    $enabled\_transitions \leftarrow find\_enabled\_transitions(model, state)$;
    $chosen\_transition \leftarrow choose\_transition(enabled\_transitions)$;
    $new\_marking \leftarrow fire\_transition(model, state, chosen\_transition, chosen\_marking)$;
    $state \leftarrow add\_marking\_to\_state(state, new\_marking, chosen\_transition)$
end
```

The semantics of a Petrinets model are defined by a reachability graph, encoding all its possible execution paths. This reachability graph, which consists of a set of reachable markings, is produced by an analyser. In such a reachability graph, a marking contains a number of tokens for each place. These markings are connected: a link denotes that the destination marking can be reached from the source marking when the transition identified by the link is triggered. The pseudocode of a Petrinets analyser is shown in Algorithm 6. It starts by initializing the *state* of the reachability graph with the initial marking of the Petrinets model. Then, it continuously adds markings to the reachability graph until no more markings can be added (*i.e.*, all paths in the semantics of the Petrinets model have been explored). To add a marking to the reachability graph, the algorithm goes through a number of phases:

1. The reachability graph is queried for the set of *unexplored markings*. These markings can potentially generate a new marking by executing one of the enabled transitions that are enabled when that marking is loaded into the model.

2. One of these markings is chosen randomly.

3. The model is queried to see which transitions are enabled if the marking is loaded into the model. A transition can be triggered when all of its input places contains at least the same amount of tokens as the weight of the links connecting the places to the transition. Transitions that are already identified by an outgoing link from the loaded marking are no longer considered, as their firing would result in an identical marking.

4. One of these transitions is chosen randomly.

5. A new marking is created by firing the transition. When the transition fires, its subtracts the number of tokens from all its input places equal to the weight of the link that connects them to the transition, and adds a number of tokens in all its output places equal to the weight of the links that connects the transition to these places (resulting in a new reachable marking).

6. The new marking is added to the reachability graph and connected to the chosen marking.

The description of this analyser is naive, since the semantics of a Petrinets model are defined as its full reachability graph. With non-trivial models, this analyser might have to run for unacceptably long amounts of time, or not finish its analysis at all, since the reachability graph might be infinite. More advanced model checkers, such as SPIN [84], do not consider the full state space, but instead analyse it with regards to a given property the user wants to check. This allows the tool to direct the search for a state where the property is *not* satisfied, thus making the analysis much more efficient. In the remainder of this section, we assume a naive implementation of a full state space generator, to demonstrate how non-deterministic formalisms can be debugged. In future work, the debugging operations that are described here can be integrated in more advanced model checkers.

5.5.2 Debugging Operations

We define the following debugging operations:

- **[Continuous Operation]** executes the reachability graph construction algorithm until the full reachability graph has been generated (this is equivalent to running the analyser without debugging support).
- **[Step]** executes one analysis "step": it generates one additional reachable node in the reachability graph.
- **[Manual]** allows the user to fully control the analyser through the visual interface. The debugging-enhanced analyser will expose these phases in the algorithm and allow the user to control them:
  1. The analyser communicates all markings that are "unexplored". An unexplored marking has at least one enabled, non-explored transition. The user chooses one of these markings by communicating its identifier.
  2. When the user has chosen one of the unexplored markings, the marking is loaded into the Petrinets model and the analyser communicates which transitions are enabled (and unexplored) in that particular marking. The user chooses one of these transitions by communicating its identifier.
  3. The analyser executes the transition and updates the reachability graph.

  This mode is most interesting to go (back) to a (partially) unexplored node in the reachability graph and continue its construction from that point interactively, exploring a particular branch in the execution semantics of the system.
- **[Breakpoints]** allow the user to automatically pause the analysis when a certain condition on the state (*i.e.*, the structure of the reachability graph) is reached. From there, the user can use one of the above operations to resume the construction of the reachability graph.

5.5.3 De- and Reconstructed Simulator

The de- and reconstructed analyser, implementing the debugging operations, is presented in Figure 5.22. Our goal was to support the operations presented in the previous subsection, while retaining the original execution semantics implemented by the simulator. This means we had to introduce points at which the algorithm can be interrupted (through stepping, breakpoints, etc.), as well as provide information on the state to the user of the debugger. An important feature to support with this formalism was the manual guiding of the reachability graph construction: the user needs to be able to choose, while debugging, in which direction the graph is constructed. It consists of two main orthogonal components: `simulation_state` and `simulation_flow`.

The `simulation_state` determines in what execution mode the simulator is. It describes how the user can influence the control flow of the simulation. Two top-level modes are defined: `paused` and `running`. The `running` state has three substates: `continuous`, `manual`, and `big step`. The simulator starts in the `paused` state, and can only transition to one of the other states when a user event is received. When the simulation terminates (because the termination condition is satisfied, or due to a breakpoint), the

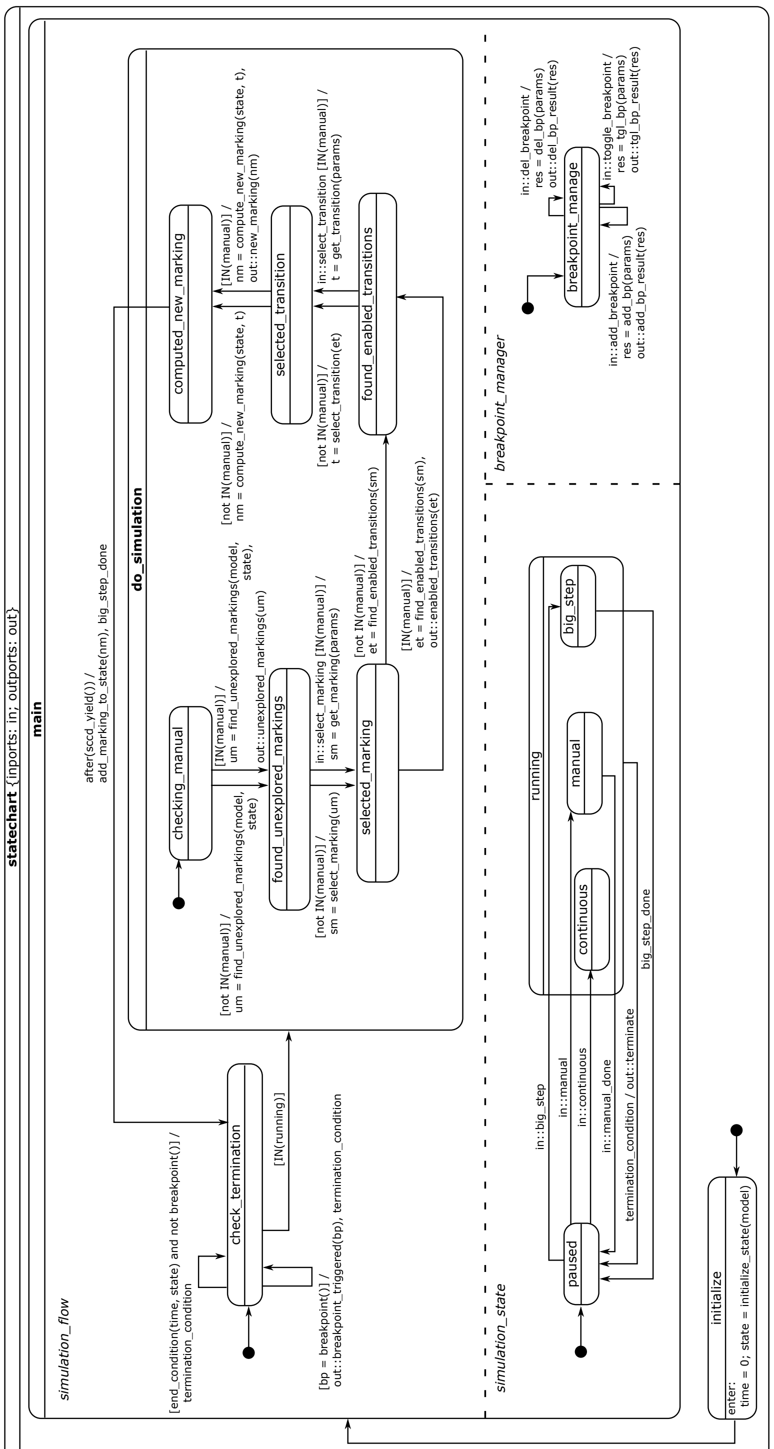

statechart {inports: in; outports: out}
main
simulation_flow
do_simulation
checking_manual
found_unexplored_markings
selected_marking
found_enabled_transitions
selected_transition
computed_new_marking
check_termination
[not IN(manual)] / um = find_unexplored_markings(model, state)
[IN(manual)] / um = find_unexplored_markings(model, state), out::unexplored_markings(um)
[not IN(manual)] / sm = select_marking(um)
in::select_marking [IN(manual)] / sm = get_marking(params)
[not IN(manual)] / et = find_enabled_transitions(sm)
[IN(manual)] / et = find_enabled_transitions(sm), out::enabled_transitions(et)
[not IN(manual)] / t = select_transition(et)
in::select_transition [IN(manual)] / t = get_transition(params)
[not IN(manual)] / nm = compute_new_marking(state, t)
[IN(manual)] / nm = compute_new_marking(state, t), out::new_marking(nm)
after(sccd_yield()) / add_marking_to_state(nm), big_step_done
[end_condition(time, state) and not breakpoint()] / termination_condition
[IN(running)]
[bp = breakpoint()] / out::breakpoint_triggered(bp), termination_condition
simulation_state
paused
running
continuous
manual
big_step
in::big_step
in::manual
in::continuous
in::manual_done
termination_condition / out::terminate
big_step_done
breakpoint_manager
breakpoint_manage
in::del_breakpoint / res = del_bp(params) out::del_bp_result(res)
in::toggle_breakpoint / res = tgl_bp(params) out::tgl_bp_result(res)
in::add_breakpoint / res = add_bp(params) out::add_bp_result(res)
initialize
enter: time = 0; state = initialize_state(model)
out::initialized(get_state())

Figure 5.22: The de- and reconstructed Petrinets analyser.

simulation state transitions to the `paused` state from any of the other states. Detecting termination is encoded in the `simulation_flow` orthogonal component, discussed below. Based on the simulation state, the behaviour of the simulation flow changes.

The `simulation_flow` orthogonal component determines the flow of the simulation steps, and explicitly encodes the invocation order of the non-modal functions. This is an explicit representation of the control flow that was implemented in the *while*-loop in the coded implementation. The orthogonal component consists of two main components: `check_termination` and `do_simulation`. In the `check_termination` state, the simulator checks whether or not simulation should continue. It has self-transitions to manage breakpoint detection, and checking whether the end condition was satisfied. Breakpoints are detected here because they need to immediately pause the simulation. If the detection were performed in a different component, the detection result would have to be communicated to the `simulation_flow` component, which might react late and already be executing the next big step. To check whether the next simulation step can be executed, the `check_termination` inspects the state of the orthogonal component `simulation_state`. If the `running` state is active, the next simulation step can be executed. The `do_simulation` composite state traverses through five stages (encoded as states), each responsible for a part of the simulation algorithm. Linking back to the original code implementation, the `check_termination` state is the condition of the *while*-loop, whereas `do_simulation` is the body of the loop. If the simulation is not in `manual` mode, the phases are executed without any communication to the user. When the simulation is in `manual` mode, however, no automatic transition takes place. The user can then manually step through the simulation phases using several operations:

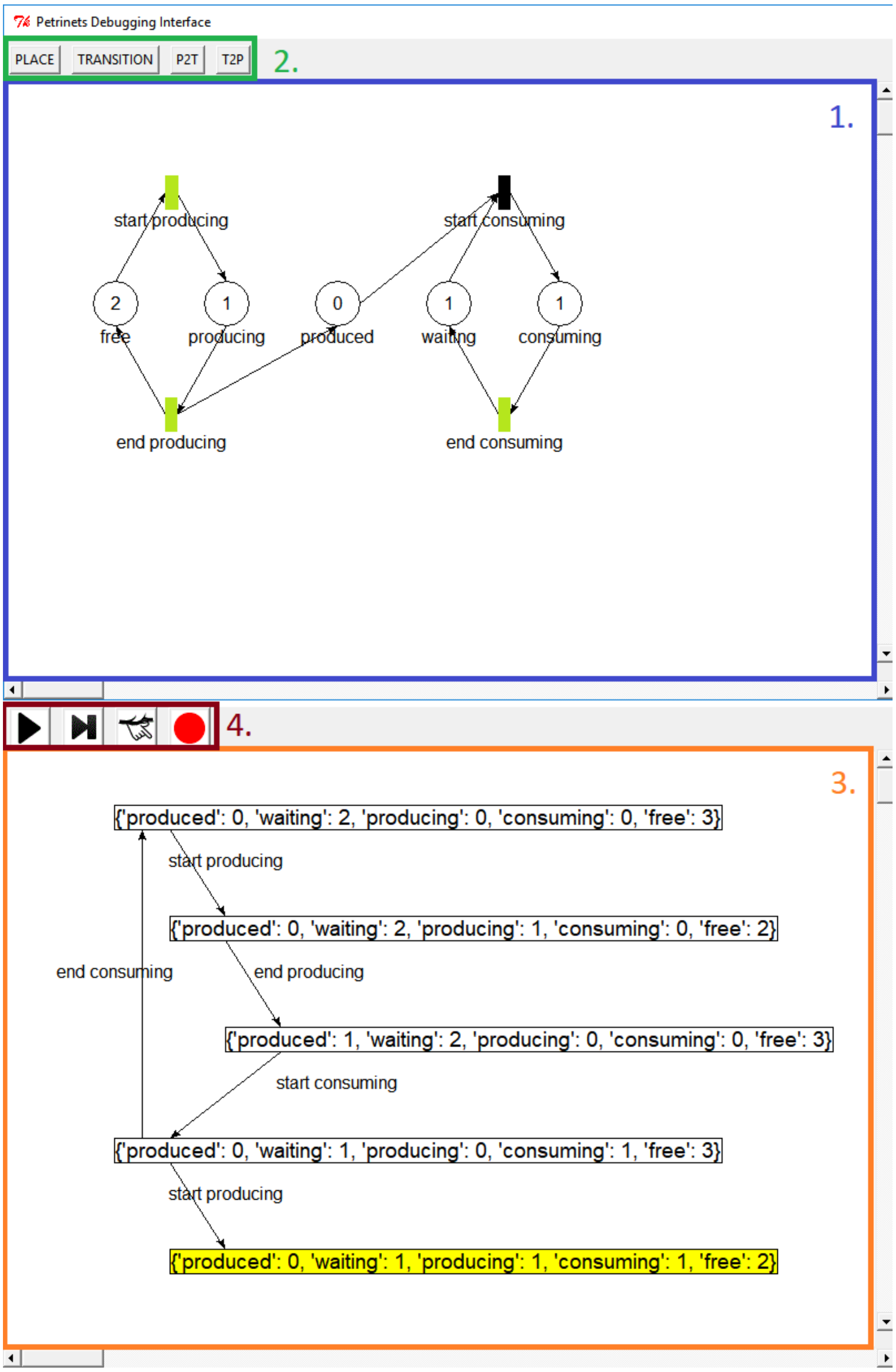
1. The user receives all unexplored markings with the *unexplored_markings* event.
2. The user chooses one unexplored marking and communicates that choice with the *select_marking* event.
3. The user receives all enabled transitions with the *enabled_transitions* event.
4. The user chooses one enabled transition and communicates that choice with the *select_transition* event.
5. The user receives the newly created marking with the *new_marking* event.

The `breakpoint_manager` can add, toggle, and delete breakpoints, as well as list all currently defined breakpoints.

Summary We demonstrated in this subsection that the debugging operations described in the previous subsection have been successfully added to the Petrinets simulator. In the next subsection, we couple the instrumented simulator to a debugging environment.

5.5.4 Debugging Environment

The visual modelling, analysis, and debugging interface is shown in Figure 5.23. At the top, the modelling interface for Petrinets is shown with a loaded model that describes a producer-consumer system **(1)**. A toolbar allows to edit the model **(2)**. Below, the (partial)



{'produced': 0, 'waiting': 2, 'producing': 0, 'consuming': 0, 'free': 3}
{'produced': 0, 'waiting': 2, 'producing': 1, 'consuming': 0, 'free': 2}
{'produced': 1, 'waiting': 2, 'producing': 0, 'consuming': 0, 'free': 3}
{'produced': 0, 'waiting': 1, 'producing': 0, 'consuming': 1, 'free': 3}
{'produced': 0, 'waiting': 1, 'producing': 1, 'consuming': 1, 'free': 2}

Figure 5.23: The interface of the Petrinets debugger.

reachability graph is shown **(3)**. A toolbar allows the user to control the analysis using the operations defined in the previous subsection **(4)**. In the screenshot, the user is manually controlling the analysis algorithm and has selected the bottom-most marking (highlighted in yellow). In the Petrinets model, the marking is loaded in the places, and the enabled transitions are highlighted in green. By clicking one of them, the transition is chosen to be fired, and a new (up to now unexplored) marking is added to the reachability graph.

Summary

This section demonstrates how debugging support can be implemented for Petrinets. The language exhibits the following semantic features of the feature diagram presented in Figure 4.2:

- It is **non-deterministic**.
- It is **untimed**.
- It has **static structure**.

Its non-determinism is the most prominent feature of this formalism. In this section, we have demonstrated that our approach can support debugging for languages exhibiting this feature.

5.6 Dynamic-Structure DEVS

A number of formalisms have been proposed to model dynamic changes in the structure of systems using discrete-event abstractions [11, 196]. Dynamic Structure DEVS was introduced by Barros [10] as an extension of the DEVS formalism. Dynamic Structure DEVS is a causal, timed (discrete-event), dynamic-structure, deterministic formalism.

5.6.1 Syntax and Semantics

The syntax of Dynamic Structure DEVS is similar to Parallel DEVS, with added support for dynamically varying structure. The basic building blocks of Dynamic Structure DEVS are *atomic* models, which can be composed hierarchically in *coupled* (network) models, whose structure can change during simulation.

An *atomic* Dynamic Structure DEVS model is a structure $< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau >$, where:

$X = \{(p, v) | p \in \mathit{IPorts}, v \in X_p\}$ is the set of input ports and values. IPorts is a set of input port names. Each port p has an associated set of possible input values X_p;

S is a set of sequential states;

$Y = \{(p, v)|p \in OPorts, v \in Y_p\}$ is the set of output ports and values. *OPorts* is a set of output port names. Each port p has an associated set of possible output values Y_p;

δ_{int}: $S \to S$ is the internal transition function;

$\tau : S \to \mathbb{R}_0^+$ is the time advance function;

δ_{ext}: $Q \times X \to S$ is the external transition function, where:

$Q = \{(s, e)|s \in S, 0 \leq e \leq \tau(s)\}$ is the total state set, with e the time elapsed since the last transition.

$\lambda : S \to Y$ is the output function.

A *coupled* Dynamic Structure DEVS model is a structure $< \chi, M_\chi >$, where χ is a reference to the network executive, and M_χ its model. M_χ is an atomic Dynamic Structure DEVS model $< X_\chi, S_\chi, Y_\chi, \delta_{int_\chi}, \delta_{ext_\chi}, \lambda_\chi, \tau_\chi >$. A state $s_\chi \in S_\chi$ is defined by the tuple $(X_\Delta, Y_\Delta, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, \Xi, \theta)$, where:

Δ is the Dynamic Structure DEVS dynamic structure network;

$X_\Delta = \{(p, v)|p \in IPorts, v \in X_p\}$ is the set of input ports and values of the Dynamic Structure DEVS dynamic structure network. *IPorts* is a set of input port names. Each port p has an associated set of possible input values X_p;

$Y_\Delta = \{(p, v)|p \in OPorts, v \in Y_p\}$ is the set of output ports and values of the Dynamic Structure DEVS dynamic structure network. *OPorts* is a set of output port names. Each port p has an associated set of possible output values Y_p;

D is a set of component references ($\chi \notin D$);

M_i is the Dynamic Structure DEVS atomic model of component i, for all $i \in D$;

I_i is the set of influencees of component i, for all $i \in D \cup \{\chi, \Delta\}$ (with $i \notin I_i$);

$Z_{i,j}$ is the transfer function, for all $i \in D \cup \{\chi, \Delta\}$, and for all $j \in I_i$, where:

$Z_{\Delta,j}$: $X_\Delta \to X_j$;

$Z_{i,\Delta}$: $Y_i \to Y_\Delta$;

$Z_{i,j}$: $Y_i \to X_j$;

$Z_{k,\chi}(y) \neq \phi \Rightarrow Z_{k,j}(y) = \phi$, for $k \in D \cup \{\Delta\}$, $y \in Y_k$, and for all $j \in I_k \backslash \{\chi\}$ with ϕ the non-event.

Ξ: $\Pi \to D \cup \{\chi\}$ is the select function, with $\Pi = 2^{D \cup \chi} \backslash \{\}$ and $\Xi(A) \in A$;

θ are other state variables not defined before.

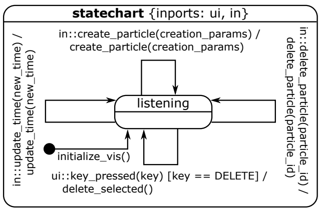
For a Dynamic Structure DEVS to be *legitimate*, only a finite number of events can differ from the non-event ϕ in any finite time interval. Dynamic Structure DEVS is closed under coupling: each coupled Dynamic Structure DEVS model can be transformed into a behaviourally equivalent atomic Dynamic Structure DEVS model. This allows nesting of Dynamic Structure DEVS coupled models to arbitrary depth. Intuitively, Dynamic Structure DEVS allows to model DEVS networks whose structure can change over time. Conceptually, the network executive is responsible for controlling these changes, by changing its state when receiving input events. Changes can be made to:

- The input and output sets $X_\triangle$ and $Y_\triangle$, by, for example, removing or adding ports.
- The set of component references D, by removing or adding components.
- The set of component models $\{M_i | i \in D\}$, by updating a component model with a new version. The structure of atomic models (including its input and output sets X and Y, state set S, transition functions δ_{int} and δ_{ext}, output function λ, and time advance function τ) can thus be changed.
- The set of influencees $\{I_i | i \in D\}$ by updating, adding, or deleting connections between components over which events can be sent or received.
- The set of transfer functions $Z_{i,j}$.

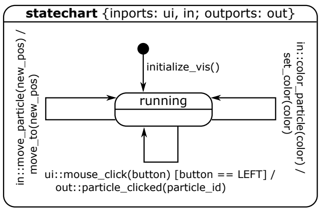
As an example, we model a simple particle interaction simulation, in which a number of particles move in a constrained space, bouncing off walls and each other when they collide. The user can interact with the simulation by sending events on the input port of the Dynamic Structure DEVS model.

Below, we list the requirements of the simulation system as they are implemented:

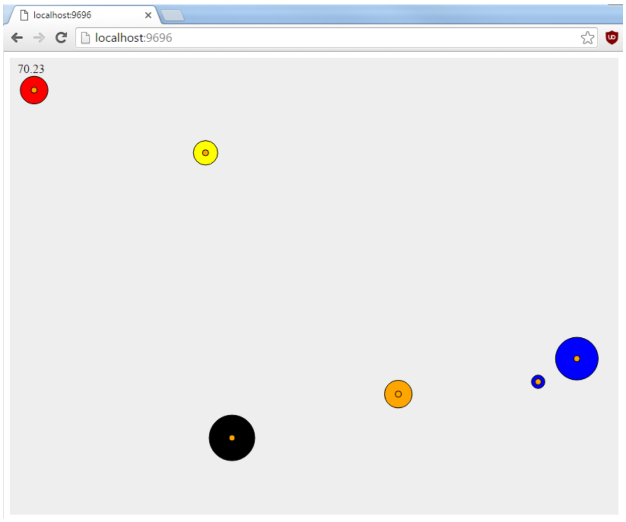
- A field holds a number of moving particles at any point in time. It has a width w and height h defining the boundaries that constrain the movement of the particles, with the top-left corner at $(0, 0)$ and the bottom-right corner at (w, h).
- A new particle is created every second.
- When a particle is created, it chooses a random radius $r \in]5.0, 30.0[$, a random (2D) position within the boundaries of the field ((x, y) with $x \in]r, w-r[$ and $y \in]r, h-r[$), and a random velocity $(v_x, v_y) \in ([-75.0, 75.0], [-75.0, 75.0])$. Its colour is set to red.
- At each frame, a particle updates its current position according to its velocity.
- 2.5 seconds after it was created, a particle changes its colour to black.
- At a random point in time, each particle selects itself for deletion, which results in its colour changing to yellow, its velocity to 0, and it being deleted after 2.5 seconds.
- When two particles collide, their colour changes to blue, and a new particle is created. The colliding particles swap their velocity vectors (bouncing off each other). The particles request, with a certain probability, for a new particle to be created.
- When a particle bounces against one of the sides of the field, it negates the normal component of its velocity to move in the other direction.
- When a particle receives an event corresponding to a user clicking on it, its colour is changed to orange and its velocity to 0. The particle becomes "selected".
- When a previously selected particle receives a "delete" event, it is deleted.


statechart {inports: ui, in}
in::create_particle(creation_params) / create_particle(creation_params)
listening
in::update_time(new_time) / update_time(new_time)
in::delete_particle(particle_id) / delete_particle(particle_id)
initialize_vis()
ui::key_pressed(key) [key == DELETE] / delete_selected()

(a) The Statecharts model of the window.


statechart {inports: ui, in; outports: out}
initialize_vis()
running
in::move_particle(new_pos) / move_to(new_pos)
in::color_particle(color) / set_color(color)
ui::mouse_click(button) [button == LEFT] / out::particle_clicked(particle_id)

(b) The Statecharts model of a particle's visualization.



(c) A screenshot of the model-specific visualization interface.

Figure 5.24: The model-specific visualization UI—models and generated application.

This system clearly exhibits dynamic structure behaviour, and the Dynamic Structure DEVS formalism is an appropriate choice to represent its dynamic-structure behaviour. Our model consists of one top-level coupled model called *Field*, whose set of input ports $IPorts = \{INTERRUPT\}$, which the user can send events to (such as "select", "delete") and set of output ports $OPorts = \{POS\_OUT, COLOR\_OUT, TIME\_OUT\}$, which are used to communicate state updates to the user during simulation. At the start of simulation, the Field creates two atomic models:

- The *PositionManager* is responsible for keeping track of the positions of each particle and detecting collisions. It receives the positions of particles when they are updated, and has an output port for each of the particles (resulting in a dedicated channel to communicate collisions detected to only the involved particles). Its set of input ports $IPorts = \{INTERRUPT, POS\_IN\}$ and its initial set of output ports $OPorts = \{TIME\_OUT\}$.

- The *ParticleSpawner* creates a new particle each second. It also has an input port to which particles can send a request to create a new particle.

A particle has a set of input ports $IPorts = \{COLLISION\_DETECT\}$ (which is connected to the dedicated output port of the PositionManager) and a set of output ports $OPorts = \{SPAWNER\_COMM, POS\_OUT, COLOR\_OUT\}$. The first is used to communicate with the ParticleSpawner, the last two to communicate the position and colour of the particle to the PositionManager and the user.

We show a suitable model-specific visualization in Figure 5.24, whose behaviour is specified using Statecharts. Figure 5.24a shows the Statecharts model of the top-level window, which receives events generated by the simulation model (*i.e.*, position and colour updates, as well as creation and deletion of particles) and user events (*i.e.*, mouse clicks and key presses) that are translated to output events. Each particle's visualization behaviour is controlled by a separate Statecharts model (shown in Figure 5.24b), created by the window when the particle is created.

An abstract simulator for a parallel version of Dynamic Structure DEVS was defined in [12]. An efficient version of the simulation algorithm was introduced in [143] and was implemented in a number of simulation tools, including *adevs* [148] and *PythonPDEVS* [204, 206].

ALGORITHM 7: The Dynamic Structure DEVS simulator's "main loop".

```
Input: Model to simulate ($model$)
1  $flatten\_model(model)$;
2  $time \leftarrow 0$;
3  $state \leftarrow initialize\_state(model)$;
4  while not end_condition(time, state) do
5      $imminents \leftarrow compute\_imminents(state)$;
6      $selected\_component \leftarrow select\_imminent(imminents)$;
7      $events \leftarrow compute\_output(selected\_component)$;
8      $route\_events(events)$;
9      $ext \leftarrow compute\_externals(state, events)$;
10     for component in imminents $\cup$ ext do
11         $compute\_next\_state(component, state)$;
12     end
13     for component in imminents $\cup$ ext do
14         $compute\_new\_structure(component, state)$;
15     end
16     $time \leftarrow time\_next(state)$;
17 end
```

The pseudo-code of a possible simulation algorithm for Dynamic Structure DEVS is shown in Algorithm 7. For efficiency reasons, the model is flattened (line 1), which avoids the use of simulators for atomic models that need to be coordinated by simulators for coupled models. The algorithm initializes the state and the clock of the simulation on line 3 and then simulates the model, stepping through simulated time, until the simulation experiment's end condition is satisfied. One iteration of the while loop (lines 4–17) is a simulation "step" and consists of eight phases:

1. [**Line 5**] Compute the imminent components (*i.e.*, the components that have their internal transition function scheduled at the current simulated time);

2. [**Line 6**] Select one imminent component;

3. [**Line 7**] Compute the output events of the imminent component;

4. [**Line 8**] Route the events along port connections, executing their transfer function;

5. [**Line 9**] Compute the set of components that will execute their external transition function as they react to incoming events;

6. [**Lines 10-12**] Compute the state of all transitioning components in arbitrary order;

7. [**Lines 13-15**] Execute a "model transition" function for each transitioning component. This function can alter the component's structure, but not its state. Model transitions are executed up the model's hierarchy until the root model is reached.

8. [**Line 16**] Compute the next simulated time value. The new value depends on the transition function(s) invoked: in case of an internal transition, the τ function of the transition component is invoked. In case of an external (realtime) interrupt, the time at which the interrupt took place needs to be taken into account.

5.6.2 Debugging Operations

We define the following debugging operations for Dynamic Structure DEVS, similar to the ones defined for Parallel DEVS in Section 5.3:

- **[As-Fast-as-Possible Simulation]** In this mode, the simulation runs as fast as the underlying hardware can manage and the operating system will allow, until the end condition is satisfied. It is comparable to running program code, which is always run as fast as possible. It is supported by most simulation kernels, as a user of the kernel generally wants to run the simulation without seeing intermediate results. At the end, the user can inspect the generated trace and any metrics collected.

- **[Real-Time Simulation]** For timed models, which, when deployed on "target" hardware, will run in real-time, simulating the system in *real-time* is a useful feature. In that case, simulated time is synchronized with the wall-clock time. For debugging purposes, a scale factor can be applied to speed up or slow down simulation, while retaining the *linear* relation between simulated time and wall-clock time. A scale factor of 1 corresponds to real-time, while a scale factor smaller or greater than 1 slows down or speeds up simulation proportionally.

- **[Pause]** Pausing a simulation allows to inspect the current state of the system and enables other debugging operations, such as stepping and state modifications.

- **[Big Step and Small Step]** To transpose the notion of stepping, we turn to the simulation algorithm shown in Algorithm 7. In the case of Dynamic Structure DEVS, we implement two types of steps: a "big step" executes one simulation step (corresponding to an iteration of the while-loop). Inside of the while-loop, a number of simulation phases, or "small steps", are executed. Providing control and feedback at this level allows the user to gain more insight into the detailed model semantics.

- **[God Event]** A "god event" allows a user to change the value of a state variable of a particular atomic Dynamic Structure DEVS model. This operation changes the current state $(s_{i,curr}, e_{i,curr}) \in Q_i$ to a new state $(s_{i,new}, 0) \in Q_i$ where Q_i is the total state set of the atomic Dynamic Structure DEVS model i. This change is performed by an "outside force" not present in the original dynamics of the model, hence the name.

- **[Event Injection]** A user can schedule to inject an event $(x, p) \in X_i$ at a specified (future) simulated time instant t. Once time t is reached, the atomic Dynamic

Structural Change \ Debugging Operation	(1) Pause	(2) Big Step	(3) Small Step	(4) Reset	(5) God Event	(6) Inject Event	(7) Breakpoint	(8) Visualization
Add or Remove State Variable	Y	Y	Y	Y	Y	Y	N	N
Add or Remove Port	Y	Y	Y	Y	Y	N	N	N
Add or Remove Connection	Y	Y	Y	Y	Y	Y	N	N
Add or Remove Component	Y	Y	Y	Y	Y	N	N	N
Change Transition Function	Y	Y	Y	Y	Y	Y	Y	N
Change Output Function	Y	Y	Y	Y	Y	Y	Y	N
Change Time Advance Function	Y	Y	Y	Y	Y	Y	Y	N
Change Transfer Function	Y	Y	Y	Y	Y	Y	Y	N
Change Select Function	Y	Y	Y	Y	Y	Y	Y	N

Table 5.1: Structural changes and their effect on debugging.

Structure DEVS model i receives the input event x on its port p, triggering its external transition function.

- **[Breakpoints]** A breakpoint is used in code debugging to pause the execution of the program when a specific line of code is reached, and when its associated (optional) condition is satisfied. As the program counter is part of the execution state of the program, a breakpoint pauses the program automatically when a state condition is satisfied. We transpose breakpoints to the Dynamic Structure DEVS formalism by allowing the user to specify conditions on the execution state of the Dynamic Structure DEVS simulation (including access to time and current (total) state information). In contrast to code debugging, we do not allow the simulation to break on a specific line of code. Instead, the simulation can be paused automatically when a condition on the simulation state is satisfied. A breakpoint is a function that returns *True* when the simulation should pause, *False* in all other cases, and it receives five parameters:

  1. The current simulation time;
  2. The current state of the simulation system, which is an aggregation of the states of its atomic Dynamic Structure DEVS components, as well as the current structure of the system;
  3. The names of atomic Dynamic Structure DEVS models that executed a transition function in the iteration preceding the triggering of the breakpoint;
  4. The output generated by the atomic Dynamic Structure DEVS models that executed their internal transition function in the iteration preceding the triggering of the breakpoint;
  5. The input received by the atomic Dynamic Structure DEVS models that executed their external transition function in the iteration preceding the triggering of the breakpoint.

  When a breakpoint is triggered, these values are passed to the user in order to help the user understand why the breakpoint was triggered.

As Dynamic Structure DEVS allows the structure of the modelled system to change during simulation, certain debugging operations that were defined for Parallel DEVS, which relied on the static structure of the system, might become meaningless. In Table 5.1, we identify the debugging operation that must be changed to accommodate Dynamic Structure DEVS. The first column lists all possible changes to the structure that are allowed by the Dynamic Structure DEVS formalism. Each column corresponds to a debugging operation. Each cell is either green (**Y**), meaning the operation is never affected by the structure change, or red (**N**), meaning the operation can be affected by the structure change. A number of debugging operations are unaffected:

(1) **[Pause]** Pausing a simulation is independent of structural changes. It pauses the simulation as soon as possible, either after the currently executing big step, or, in the case of real-time simulation, potentially during a waiting period. This ensures the system is in a stable (or quiescent) state that can be inspected by the user.

(2)-(3) **[Big Step and Small Step]** Stepping through a simulation is similarly unaffected. It can be compared to manually pausing after each iteration, or after the execution of a simulation phase. An additional small step (corresponding to lines 13-15 in Algorithm 7) was added for communicating the structural changes to the user, but the functioning of the small step operation is not affected by changes in the system structure during simulation.

(4) **[Reset]** Resetting the simulation restores the initial state of the system.

(5) **[God Event]** A god event changes the value of a state variable. While components and their variables can be added and/or removed during simulation, a god event is only accepted when the simulation is paused. At that moment, the user is aware of the current structure of the system, and is only allowed to change the value of instance variables that exist.

In contrast, three debugging operations are affected, and will have to implement additional checks to ensure they still function as intended:

(6) **[Event Injection]** A user can inject an event on an input port at a specified point in (future) simulation time. Two changes have an effect on this operation: when a component or a port that is the recipient of an injected event is removed before the injection time is reached, that injection becomes invalid. We solve this by ignoring the injected event in case the input port was not found.

(7) **[Breakpoints]** When modelling a breakpoint, the user is not aware of future structural changes. The breakpoint might attempt to access a component, its ports, or its variables that in the meantime are removed. We disable the breakpoint automatically when it tries accessing a non-existent component and warning the user.

(8) **[Visualization]** As the visualization is responsible for displaying the state of the system, and that state now contains the structure of the system, it is affected by all operations that change the structure of the system. The visualization of the state has to be appropriately updated when components, variables, ports, components (and their functions) are added, removed, or changed. The point in time when the visualization is updated depends on the mode of simulation. In as-fast-as-possible simulation, the visualization is updated at the end of simulation. In real-time simulation and while

the user is big stepping, the visualization is updated after each big step. When the user is small stepping, the visualization is updated after each small step. Although not every small step has an effect on the state of the system, additional information concerning the simulation algorithm is displayed.

5.6.3 De- and Reconstructed Simulator

The debugging-enhanced simulator for the Dynamic Structure DEVS formalism is shown in Figure 5.25. Our goal was to support the operations presented in the previous subsection, while retaining the original execution semantics implemented by the simulator. This means we had to introduce points at which the algorithm can be interrupted (through stepping, breakpoints, etc.), as well as provide information on the state to the user of the debugger. Important in this formalism is its support for dynamic structure, which requires ways of communicating the structural changes that occur after a "big step" has ended to the user of the debugger. The Statecharts model accepts (user) events on an input port *in* and communicates the simulation results on an output port *out*. It consists of two main orthogonal components: `simulation_state` and `simulation_flow`.

The `simulation_state` determines in what execution mode the simulator is. It describes how the user can influence the control flow of the simulation. Four modes are defined: `paused`, `continuous`, `realtime`, and `big_step`. The simulator starts in the `paused` state, and can only transition to one of the other states when a user event is received. Pausing is only possible when running the simulation in continuous mode or in realtime mode. The debugging-enhanced simulator does not allow pausing the simulation when computing a big step or small step, as that could leave the simulator in a (globally) inconsistent state. Rather, the simulator pauses automatically after every big step when in big step mode, and small steps are manually controlled (discussed below). When the simulation terminates (because the termination condition is satisfied, or due to a breakpoint), the simulation state transitions to the `paused` state from any of the other states. Detecting termination is encoded in the `simulation_flow` orthogonal component, discussed below. Based on the simulation state, the behaviour of the simulation flow changes.

The `simulation_flow` orthogonal component determines the flow of the simulation steps, and explicitly encodes the invocation order of the non-modal functions. This is an explicit representation of the control flow that was implemented in the *while*-loop in the coded implementation. The orthogonal component consists of two main components: `checking_termination` and `do_simulation`. In the `checking_termination` state, the simulator checks whether or not simulation should continue. It has self-transitions to manage god events (manual state changes), breakpoint detection, and checking whether the end condition was satisfied. By defining these events to only be processed (using transitions) in the `checking_termination` state, we ensure god events are only possible when a simulation "big step" is currently not executing. Breakpoints are detected here because they need to immediately pause the simulation. If the detection were performed in a different component, the detection result would have to be communicated to the `simulation_flow` component, which might react late and already be executing the next big step. To check whether the next simulation step can be executed, the `checking_termination` inspects the state of the orthogonal component `simulation_state`, and, if realtime simulation is selected, also checks whether the

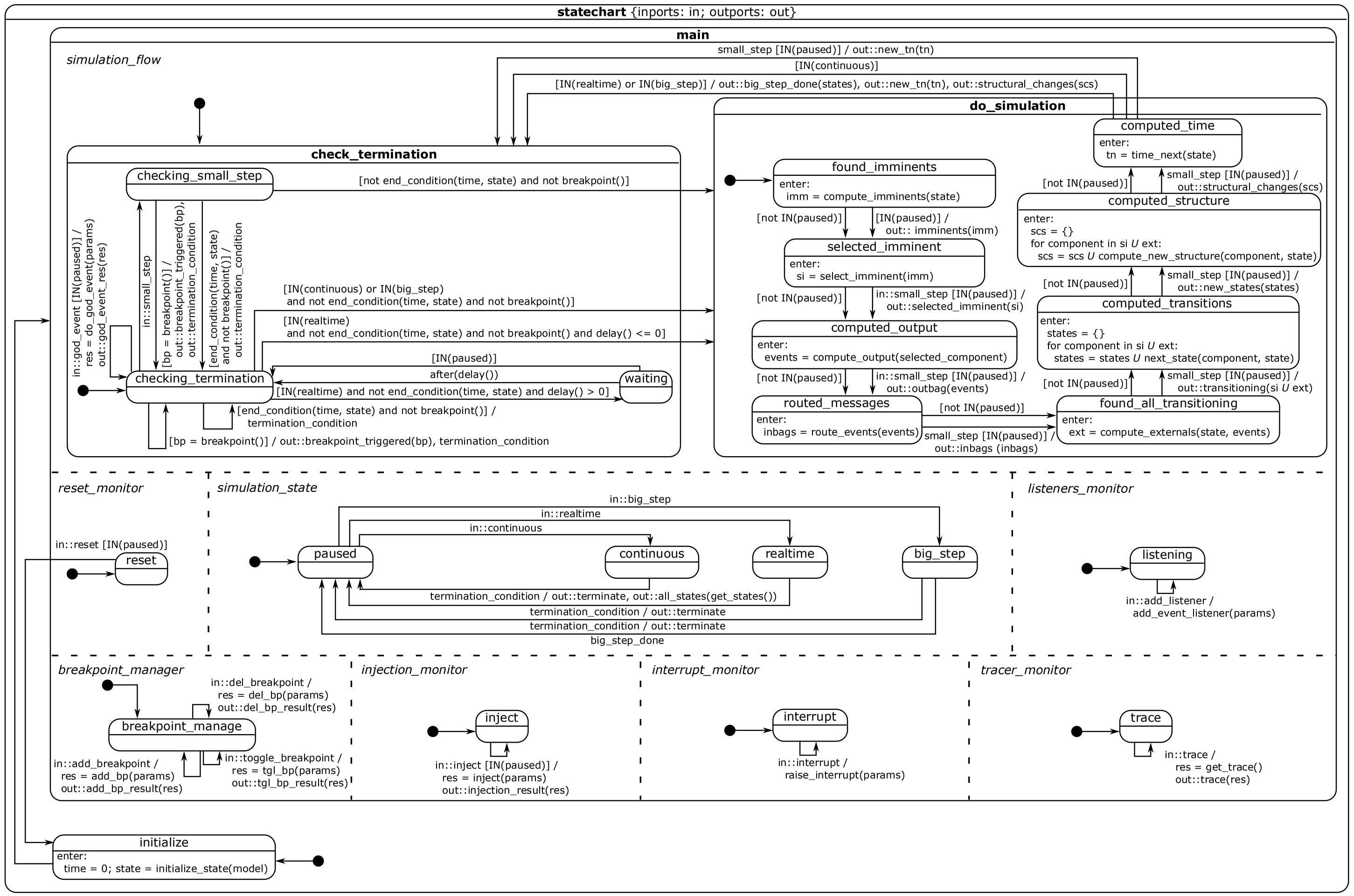

statechart {inports: in; outports: out}
main
simulation_flow
small_step [IN(paused)] / out::new_tn(tn)
[IN(continuous)]
[IN(realtime) or IN(big_step)] / out::big_step_done(states), out::new_tn(tn), out::structural_changes(scs)
check_termination
checking_small_step
[not end_condition(time, state) and not breakpoint()]
in::god_event [IN(paused)] / res = do_god_event(params) out::god_event_res(res)
in::small_step
[bp = breakpoint()] / out::breakpoint_triggered(bp), out::termination_condition
[end_condition(time, state) and not breakpoint()] / out::termination_condition
[IN(continuous) or IN(big_step) and not end_condition(time, state) and not breakpoint()]
[IN(realtime) and not end_condition(time, state) and not breakpoint() and delay() <= 0]
[IN(paused)]
after(delay())
checking_termination
waiting
[IN(realtime) and not end_condition(time, state) and delay() > 0]
[end_condition(time, state) and not breakpoint()] / termination_condition
[bp = breakpoint()] / out::breakpoint_triggered(bp), termination_condition
do_simulation
computed_time
enter: tn = time_next(state)
found_imminents
enter: imm = compute_imminents(state)
[not IN(paused)]
[IN(paused)] / out:: imminents(imm)
selected_imminent
enter: si = select_imminent(imm)
in::small_step [IN(paused)] / out::selected_imminent(si)
computed_output
enter: events = compute_output(selected_component)
in::small_step [IN(paused)] / out::outbag(events)
routed_messages
enter: inbags = route_events(events)
small_step [IN(paused)] / out::inbags (inbags)
found_all_transitioning
enter: ext = compute_externals(state, events)
small_step [IN(paused)] / out::transitioning(si U ext)
computed_transitions
enter: states = {} for component in si U ext: states = states U next_state(component, state)
small_step [IN(paused)] / out::new_states(states)
computed_structure
enter: scs = {} for component in si U ext: scs = scs U compute_new_structure(component, state)
small_step [IN(paused)] / out::structural_changes(scs)
reset_monitor
in::reset [IN(paused)]
reset
simulation_state
in::big_step
in::realtime
in::continuous
paused
continuous
realtime
big_step
termination_condition / out::terminate, out::all_states(get_states())
termination_condition / out::terminate
termination_condition / out::terminate
big_step_done
listeners_monitor
listening
in::add_listener / add_event_listener(params)
breakpoint_manager
breakpoint_manage
in::del_breakpoint / res = del_bp(params) out::del_bp_result(res)
in::add_breakpoint / res = add_bp(params) out::add_bp_result(res)
in::toggle_breakpoint / res = tgl_bp(params) out::tgl_bp_result(res)
injection_monitor
inject
in::inject [IN(paused)] / res = inject(params) out::injection_result(res)
interrupt_monitor
interrupt
in::interrupt / raise_interrupt(params)
tracer_monitor
trace
in::trace / res = get_trace() out::trace(res)
initialize
enter: time = 0; state = initialize_state(model)

Figure 5.25: The de- and reconstructed Dynamic Structure DEVS simulator.

next simulation step can already be performed, or if the simulation algorithm needs to wait. In case we should wait for the next simulation step, the `waiting` state is entered. Otherwise, the `do_simulation` state is entered. If the simulation is executing in real time and the delay has not passed, the simulator has to wait. Control is passed to the `waiting` state, and a transition is scheduled to fire after the delay has passed. But, pausing needs to be possible while waiting. So, a transition is added which lets the control go back to the `check_termination` state when the user requests a pause. In any other cases, the next simulation step can be executed. The `do_simulation` composite state traverses through seven stages (encoded as states), each responsible for a part of the simulation algorithm. Linking back to the original code implementation, the `check_termination` state implements the condition of the *while*-loop, whereas `do_simulation` implements the body of the loop. If the simulation is not paused, the phases are executed without any communication to the user. When the simulation is paused, however, no automatic transition takes place. The user can then manually step through the simulation phases using the "small step" operation. After every small step, relevant information is sent to the user to be displayed in a debugging interface. The first small step is requested when the `simulation_flow` is in `checking_termination`. In that case, there are two possibilities: either the simulation is paused, or it has terminated. To detect these cases, the system transitions to a `checking_small_step` state when the first small step is requested. Only if the end condition is not satisfied and no breakpoints trigger is the first phase executed. At the end of the `do_simulation` state, there are three possibilities, each with a different response to the user. If simulation is running in realtime or a big step was being executed, all information is passed to the user. If the current simulation iteration is ended by a small step, only a part of the information needs to be sent; all other information has already been sent during previous steps. If simulation is running as fast as possible, no information is sent to the user, as it would only slow down simulation.

Six additional orthogonal components are modelled:

- The `reset_monitor` listens for user requests to reset the simulation, and allows it when the simulation is paused.
- The `breakpoint_manager` can add, toggle, and delete breakpoints, as well as list all currently defined breakpoints.
- The `injection_monitor` listens for user requests to inject an event on a port in the model, and allows such injections if the simulation is paused.
- The `interrupt_monitor` is used in real-time simulations for real-time interrupts from the environment.
- The `listeners_monitor` is used in real-time simulations for listening to events on an output port.
- The `tracer_monitor` can print the full textual trace of the simulation.

From this model, the code for the debugging-enhanced simulator can be generated using an appropriate Statecharts compiler.

Summary We demonstrated in this subsection that the debugging operations described in the previous subsection have been successfully added to the Dynamic Structure DEVS

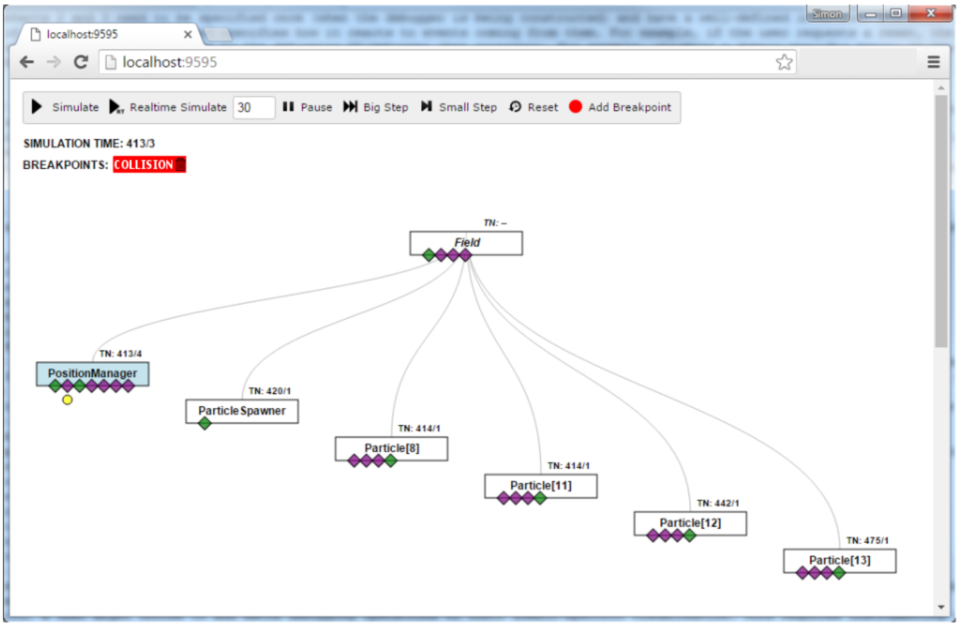


Figure 5.26: A screenshot of the debugging interface.

simulator. In the next subsection, we couple the instrumented simulator to a debugging environment.

5.6.4 Debugging Environment

Once the debugging enhanced simulation kernel is developed, an interface for interacting with it is needed. This section presents a basic visual debugging interface. The debugging interface allows for full control over the simulation algorithm using the set of debugging operations defined in the previous sections.. A screenshot of the interface is shown in Figure 5.26.

At the top, a toolbar allows the user to interact with the running simulation by pressing buttons. Below that, information related to the running simulation is visualized: the current simulated time and the current set of breakpoints are shown at the top, and below that, the current structure of the system. Each Dynamic Structure DEVS model is represented by a rectangle. The name of each component is displayed (coupled models have their names printed in italics), as well as their interface in the form of input (green) and output (purple) ports. Hovering over a port displays the name of the port as well as any incoming or outgoing connections to other ports. Hovering over (atomic) models displays their current state. The next time ("TN") at which an internal transition is scheduled is displayed above each atomic model. This is not the case for coupled models (such as *Field*): they do not have an internal transition function, and their next transition time is the minimum of its constituent components' next transition times. Clicking on an input port allows the user to inject an event. Right-clicking on an atomic model allows the user to insert a god event, which changes the value of a state variable. The hierarchical structure of the Dynamic

Structure DEVS model is displayed as a tree—children of a coupled Dynamic Structure DEVS model are shown below their parent. Models at the same level are staggered, to allow for more elements to be displayed side-by-side. Additionally, this layout helps with the drawing of incoming and outgoing connections between ports. While the example shows a two-level tree, this is not necessarily always the case: the tree can be arbitrarily deep (depending on the current model structure).

When the user executes a small step, the instrumented simulation kernel communicates useful information regarding the executed phase. The debugging interface then visualizes this information. This visualization, including an explanation of each small step, is shown in Table 5.2. Small steps allow the user to debug the system at a more fine-grained level, inspecting more closely what happens during a simulation step.

The behaviour of the debugging interface is modelled using Statecharts as well. Each user action corresponds to an input event to this model and is translated to an output event that is sent to the instrumented simulator. Conversely, any events sent from the instrumented simulator serve as input to the Statecharts model of the debugging interface.

We observe that the visualization previously defined for the simulation system is a good candidate for instrumentation. We instrument the models in Figure 5.24a and Figure 5.24b to handle two cases:

- When the simulation is reset, the initial state of the visualization has to be restored as well. To implement this, an event handler is defined in the Statecharts model of the window, and a translation is provided between the reset output event of the debugging UI and the newly defined input event of the model-specific visualization UI.

- When the simulation is paused, the domain expert might want to find out which atomic DEVS model in the debugging UI (see Figure 5.26) corresponds to a particle on the screen. To implement this, an event handler is defined in the particle Statecharts model, to raise an output event when the user right-clicks. A translation is provided between this newly defined output event and an input event of the debugging UI which results in showing the state of the atomic model corresponding to the particle.

Summary

This section demonstrates how debugging support can be implemented for Dynamic Structure DEVS. The language exhibits the following semantic features of the feature diagram presented in Figure 4.2:

- It has **dynamic structure**.

- It is **deterministic**.

- It is implements **discrete-event** semantics.

Its dynamic structure is the most prominent feature of this formalism. In this section, we have demonstrated that our approach can support debugging for languages exhibiting this feature.

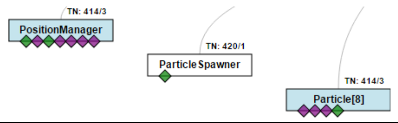
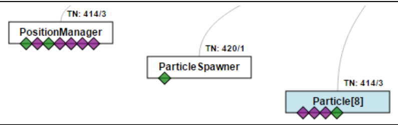
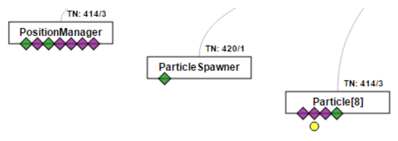
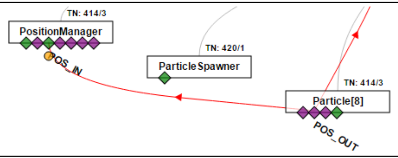
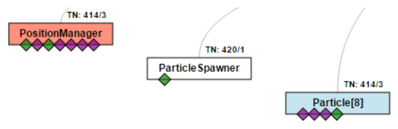
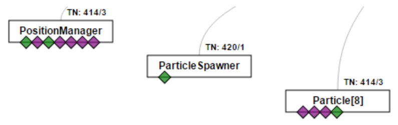
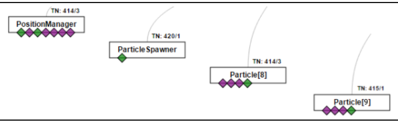
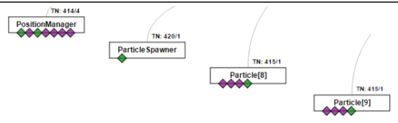
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(1) All imminent components are highlighted in blue.
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(2) Only the selected imminent component is highlighted in blue.
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(3) The output message(s) generated by the imminent component are shown beneath the corresponding port(s). Clicking the message will log its contents in the console.
 TN: 414/3 PositionManager POS_IN TN: 420/1 ParticleSpawner TN: 414/3 Particle[8] POS_OUT	(4) The message is routed from the output port(s) to the input port(s) along the connections, and translated in the process.
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(5) All transitioning components are highlighted: red for components that will execute their external transition function, blue for those that will execute their internal transition function.
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(6) The new state of the components is computed (no visual change, but hovering over the atomic models will show their new state).
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8] TN: 415/1 Particle[9]	(7) Structural changes are displayed: an atomic model was created.
 TN: 414/4 PositionManager TN: 420/1 ParticleSpawner TN: 415/1 Particle[8] TN: 415/1 Particle[9]	(8) The time at which the next internal transition function is scheduled is updated for each component.

Table 5.2: Small step visualization: the eight phases of a simulation iteration. These correspond to the steps of the simulation algorithm in Algorithm 7 (lines 5– 16).

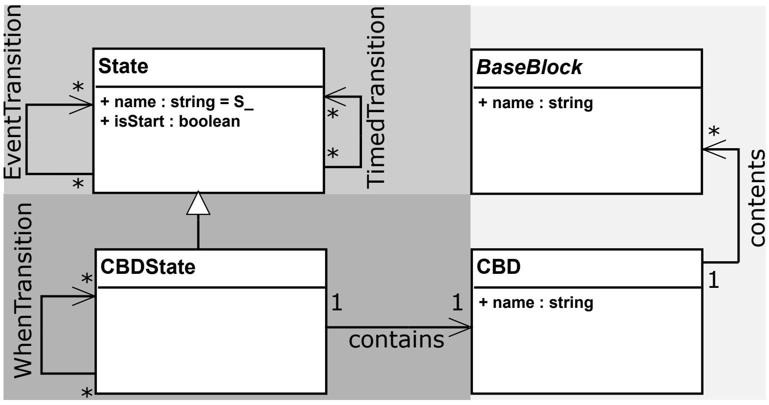


Figure 5.27: The metamodel for the Hybrid Automata language.

5.7 Hybrid TFSA-CBD

In order to observe the global behaviour of systems that exhibit both discrete and continuous behaviour, the system's behavioural models that describe the continuous and discrete parts of the behaviour are coupled. Since they conform to different formalisms, however, their interaction needs to be determined. It can consist of input/output variables that can be accessed by both models during their coupled simulation—as in co-simulation [65], or they can be more complex, realized through *boundary concepts* that do not belong to any of the coupled formalisms—as in co-modelling [18, 86]. We focus on the latter, and present a combination of Timed Finite State Automata with Causal Block Diagrams (from now on called Hybrid Automata). To debug this language, we need to combine debugging support for the two individual languages, as well as take into account the boundary concepts defined in the combination. Hybrid Automata is a causal, timed (discrete-event), hybrid (embedded), static-structure, deterministic formalism.

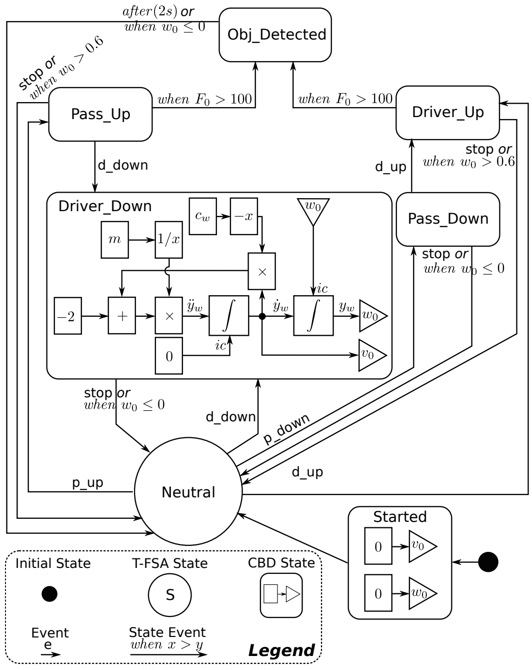
5.7.1 Syntax and Semantics

The Hybrid Automata formalism inherits its syntax from both T-FSA and CBD. Its metamodel is shown in Figure 5.27, and is divided into three parts:

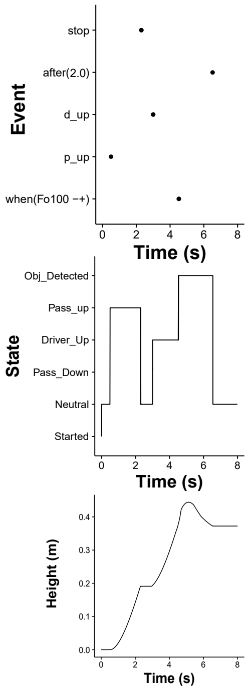
1. The syntax of T-FSA (upper-left part of the figure) has one class: the *State*. A state has a unique name, and one state of the model is the initial state. States are connected by transitions, which model the dynamics of the system. Two types of transition are defined: the *TimedTransition*, which is triggered by a timeout, and the *EventTransition*, which is triggered by an event coming from the environment.

2. The syntax of CBD (right part of the figure), which has a CBD class which contains a number of uniquely named blocks. For brevity, we have not included the subclasses of the *BaseBlock* class. The full CBD metamodel was presented in Figure 5.3.

3. A "glue" syntax that connects the two metamodels and defines the boundary concepts. A *CBDState* class is defined as a subclass of the T-FSA *State* class. Such a state con-



(a) Example hybrid model of the power window system.



(b) Example trace.

Figure 5.28: The example model and its simulation trace.

tains a full CBD model. An additional transition type is defined: the *WhenTransition*, triggered when a value of the CBD model crosses a boundary.

The "glue" part of the syntax is introduced to accommodate communication between the two formalisms. It is assumed here that the language engineer creating the hybrid language introduces this syntax manually, and also defines its semantics properly in the hybrid simulation algorithm (see the discussion later in this chapter). A more repeatable way of combing two languages would consider the "glue" as a third language (or language fragment), which includes its syntax and semantics. A three-way (automatic) merge of the fragments would then result in the hybrid language: both its syntax and semantics. To perform such merges automatically and properly is an area of active research [141], and we consider the integration of hybrid language engineering (using fragments) with debugging as future work.

Figure 5.28a shows the (partial) hybrid model for a power window system. The overall modes of the power window are depicted as a T-FSA. The *Driver_Down* state's CBD model is expanded, representing the dynamics of the window when going down. The model is initialized in the *Started* state, which is a *CBDState* that initializes the values for the variables v_0 and w_0. It then transitions to the T-FSA state *Neutral*, the only state in the model that does not contain a CBD model. Five events can be raised by the environment:

1. *d_up* signifies that the driver pressed the button to make the window go in the upwards direction.

2. *d_down* signifies that the driver pressed the button to make the window go in the downwards direction.
3. *p_up* signifies that the passenger pressed the button to make the window go in the upwards direction.
4. *p_down* signifies that the passenger pressed the button to make the window go in the downwards direction.
5. *stop* signifies that either the driver or the passenger has released a button.

In the model, four states are defined that correspond to the modes the window can be in, based on the events it receives from the environment:

1. *Driver_Up* models the dynamics of the system when the driver has requested the window to go in the upwards direction.
2. *Driver_Down* models the dynamics of the system when the driver has requested the window to go in the downwards direction.
3. *Pass_Up* models the dynamics of the system when the passenger has requested the window to go in the upwards direction.
4. *Pass_Down* models the dynamics of the system when the passenger has requested the window to go in the downwards direction.

If an object is detected (modelled by a transition that is triggered when the force exerted on the window crosses a boundary), the dynamics of the system are governed by the *Obj_Detected* state, where the window is lowered automatically. The variables v_0 and w_0 model the velocity and the position of the window, respectively. Multiple transitions are modelled that detect when the window has reached the *down* position ($w_0 \leq 0$) or the *up* position ($w_0 > 0.6$).

An example behavioural trace of the system (the result of simulation) is shown in Figure 5.28b. The system starts in the *Started* state and immediately transitions to the *Neutral* state autonomously. A *p_up* event is received, and the state changes to *Pass_Up*. The window starts going up, as the value of the height variable (w_0) increases. At some point, the passenger stops pressing the button, and the system returns to the *Neutral* state. The window stops moving. Some time later, a *d_up* event is received, and the window starts moving up again. At some point in time, however, a force is detected, and the system transitions to its *Obj_Detected* state. The window moves down for two seconds and then remains at its position.

The hybrid Hybrid Automata simulator is a merge of the algorithm for T-FSA presented in Algorithm 8 and the CBD simulator in Algorithm 3. Intuitively, the simulation algorithm can be broken down as follows:

- The simulation is initialized as in the T-FSA algorithm, with one difference: the δ_t parameter is set to the minimum of the δ_t parameters for the two simulators, guaranteeing the same level of accuracy as in the individual simulators.
- An outer-while loop executes the model until an end state has been reached, or if at any point, the currently executing CBD model has reached its maximum number of iterations.

ALGORITHM 8: The T-FSA simulator.

```
Input: Model to simulate ($M$), a time-annotated list of events ($evs$), a time step ($\delta_t$)
1  $time \leftarrow 0$;
2  $elapsed \leftarrow 0$;
3  $state \leftarrow initialize\_state(M)$;
4  while not end_condition(M, time, state) do
5      $continue \leftarrow True$;
6      while $continue$ do
7          $(evs, e_i) \leftarrow popEventAt(evs, t)$;
8          if $e_i = \emptyset$ then
9              $tr \leftarrow selectTimedTransition(M, state, elapsed)$;
10         else
11             $tr \leftarrow selectEventTransition(M, state, e_i)$;
12         end
13         if $tr \neq \emptyset$ then
14             $elapsed \leftarrow 0$;
15             $continue \leftarrow True$;
16             $state \leftarrow targetState(M, tr)$;
17         else
18             $continue \leftarrow False$;
19         end
20     end
21     $time \leftarrow time + \delta_t$;
22 end
```

- At the start of a big step, the algorithm checks whether the current state contains a CBD model. If this is the first time the state was entered, the model is initialized. Then, its next iteration is computed, by invoking the child CBD model's simulator.

- After the iteration is computed, the algorithm checks whether any state events occur: these are boundaries that are crossed by continuous variables in the CBD simulation. These boundary crossings are translated to T-FSA events and can trigger a transition. The naive algorithm for this boundary crossing considers that the boundary is crossed exactly following a big step. This, of course, might not be the case. State event location [152] is the general technique for computing the exact point at which the boundary was crossed. This requires to go back and forth in the simulation of the continuous model to exactly pinpoint the simulated time instant when the boundary cross occurred.

- A T-FSA small step is executed as usual: the next event is read from the environment and a transition is executed if any is enabled. Transitions can now also be triggered by state events, but events from the environment get priority.

This can be seen as a *protocol* which meaningfully combines the semantics of both simulators and is implemented by the hybrid simulator, which appropriately calls its child simulators. The implementation of this protocol is made possible only when the child simulators' interface provides adequate control. A more detailed discussion of language and simulator composition for hybrid languages can be found in [141].

5.7.2 Debugging Operations

We define the following debugging operations, divided into three categories: T-FSA debugging operations, CBD debugging operations, and Hybrid Automata operations.

Timed Finite State Automata

- **[As-Fast-as-Possible Simulation]** In this mode, the simulation runs as fast as the underlying hardware can manage and the operating system will allow, until the end condition is satisfied. It is comparable to running program code, which is always run as fast as possible. At the end, the user can inspect the generated trace and any metrics collected.
- **[Real-Time Simulation]** In this mode, simulated time is synchronized with the wall-clock time. For debugging purposes, a scale factor can be applied to speed up or slow down simulation, while retaining the *linear* relation between simulated time and wall-clock time. A scale factor of 1 corresponds to real-time, while a scale factor smaller or greater than 1 slows down or speeds up simulation proportionally. State is observed throughout real-time simulation after each iteration of the simulation loop.
- **[Pause]** Pausing a simulation allows the user to inspect the current state of the system and enables other debugging operations, such as stepping and state modifications.
- **[Big Step and Small Step]** To transpose the notion of stepping, we turn to the simulation algorithm shown in Algorithm 8. In the case of Statecharts, we implement two types of steps:
  - A *big step* executes transitions until none are enabled.
  - A *small step* executes one enabled transition.
- **[God Event]** A "god event" allows a user to change the value of a state variable of the T-FSA model's memory.
- **[Event Injection]** A user can inject an event on one of the input ports of the T-FSA model at the "current" time.
- **[Breakpoints]** A user can specify a breakpoint in the form of a condition on the execution state of the T-FSA simulation (including simulated time and current (total) state information). In contrast to code debugging, we do not allow the simulation to break on a specific line of code. Instead, the simulation can be paused automatically when a condition on the simulation state is satisfied. We allow the user to scope the breakpoint, by only evaluating the breakpoint when a specified state is entered. A breakpoint is a function that returns *true* when the simulation should pause, *false* in all other cases.

Causal Block Diagrams

The CBD simulator is in control when the simulation is in a *CBDState*. The CBD simulator's semantics, and its debugging operations, were presented in Section 5.2. A CBD "big step" is executed continuously until an external event comes in, or a state condition is satisfied. State conditions are checked by the Hybrid Automata simulator, which requests the CBD simulator to execute one big step repeatedly. For debugging purposes, we define the following operations:

- **[Continuous Simulation]** runs the simulation until an external event comes in, or a state condition is satisfied.
- **[Pause]** interrupts the running simulation. In continuous simulation, the simulation stops after the currently executing "big step" has finished, ensuring the system is in a stable state.
- **[Big Step]** advances the simulation with one "big step" (one iteration of the outer *while* loop) and communicates the state of the simulation.
- **[Small Step]** advances the simulation with one "small step" (one iteration of the inner *while* loop) and communicates the state of the simulation. This allows the user to step through individual block computations.
- **[Breakpoints]** can be modelled as elements that read the output value of a block in the model. The breakpoint pauses the simulation when the output of its connected element has a non-zero value. This allows users to model arbitrary conditions (as they can be modelled as a network of CBD blocks) in the CBD model on which to pause.

Hybrid Automata

The hybrid Hybrid Automata simulator is a merge of the algorithm for T-FSA presented in Algorithm 8 and the CBD simulator in Algorithm 3. Since the hybrid formalism is obtained by embedding the CBD formalism into the T-FSA formalism, the T-FSA simulator can be seen as the "master", while the CBD simulator is the "slave". In our case, this means the T-FSA simulator is in control, and the CBD simulator is only called when needed (*i.e.*, if a T-FSA state contains a CBD model). The debugging operations at the hybrid level are identical to the T-FSA debugging operations. We add, however, a way of switching between simulators using a *step into* operation. This operation is enabled if the simulation is paused and the current T-FSA state contains a CBD model.

5.7.3 De- and Reconstructed Simulator

We introduced the simulation algorithms for the T-FSA and CBD formalisms in Algorithm 8 and Algorithm 3. At a high level of abstraction, these algorithms are very similar, as they go through a set of phases:

1. *Initialization* of the simulated time and the simulation state;
2. *Execution* of simulation 'steps' until an end condition is satisfied, which forms the core of the algorithm, where a new state is computed based on the previous one, and the simulation time is advanced;

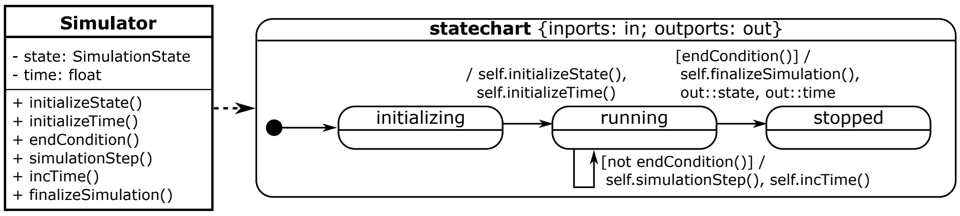

Simulator
- state: SimulationState
- time: float
+ initializeState()
+ initializeTime()
+ endCondition()
+ simulationStep()
+ incTime()
+ finalizeSimulation()
statechart {inports: in; outports: out}
/ self.initializeState(), self.initializeTime()
[endCondition()] / self.finalizeSimulation(), out::state, out::time
initializing
running
stopped
[not endCondition()] / self.simulationStep(), self.incTime()

Figure 5.29: The canonical form of the generic simulation algorithm.

3. *Finalization* where, for example, the final state of the simulation and the time at which it ended is communicated to the user.

ALGORITHM 9: Generic simulation algorithm.

```
Input: Model to simulate ($M$), list of parameters ($params$)
$initialize(params)$;
while not $endCondition()$ do
    $executeStep()$;
end
$finalize()$;
return $getState(), getTime()$
```

These phases yield a generic template, shown in Algorithm 9. Converting this high-level control flow to a SCCD model is possible using the 'de- and reconstruction' technique presented in Section 4.3 and was applied to multiple formalisms in the previous sections. The result is shown in Figure 5.29: we model a class *Simulator* which has the necessary attributes and methods (whose implementation depends on the formalism being simulated). The behaviour of this class is modelled in a Statecharts model which executes the phases of simulation outlined above.

Hierarchical Canonical Representation

For the purposes of debugging, as well as simulator composition, we need to refine the instruction $executeStep()$ in Algorithm 9. There is a single notion of 'step': it is a computation which brings the simulation from a state to the next and increases the simulated time. Upon closer inspection of Algorithm 8 and Algorithm 3, an additional level of 'steps' can be discerned. Each 'step' (from now on: 'big step') consists of a series of 'small steps'. In the case of T-FSA, a 'big step' executes as many transitions as possible, while a 'small step' executes one such transition. In the case of CBD, a 'big step' computes the new value of all signals in the model, while a 'small step' computes the value of a single signal. This means the executeStep() function can be expanded as a *while*-loop, which is preceded by an initialization phase and succeeded by a finalization phase. In [141], the authors differentiate analogously between a 'macro' and a 'micro' step, and propose a flattened canonical form of the simulator. We propose an different, hierarchical, canonical form, where each level is modelled in a separate SCCD class, following the template of Figure 5.29.

As an example, Figure 5.30 demonstrates the hierarchical canonical form for the T-FSA

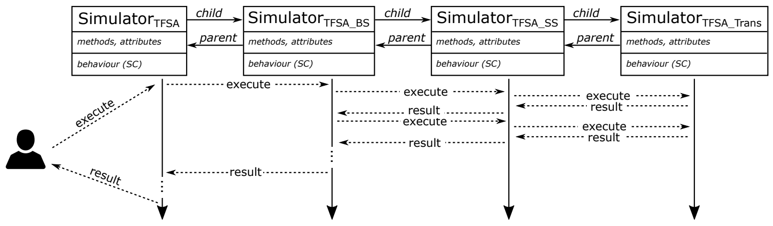


Figure 5.30: The hierarchical structure of the T-FSA simulator.

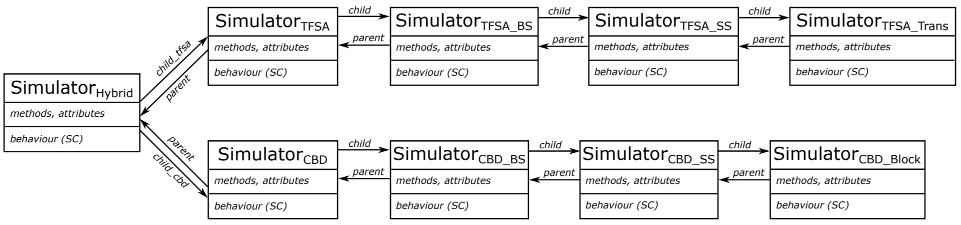


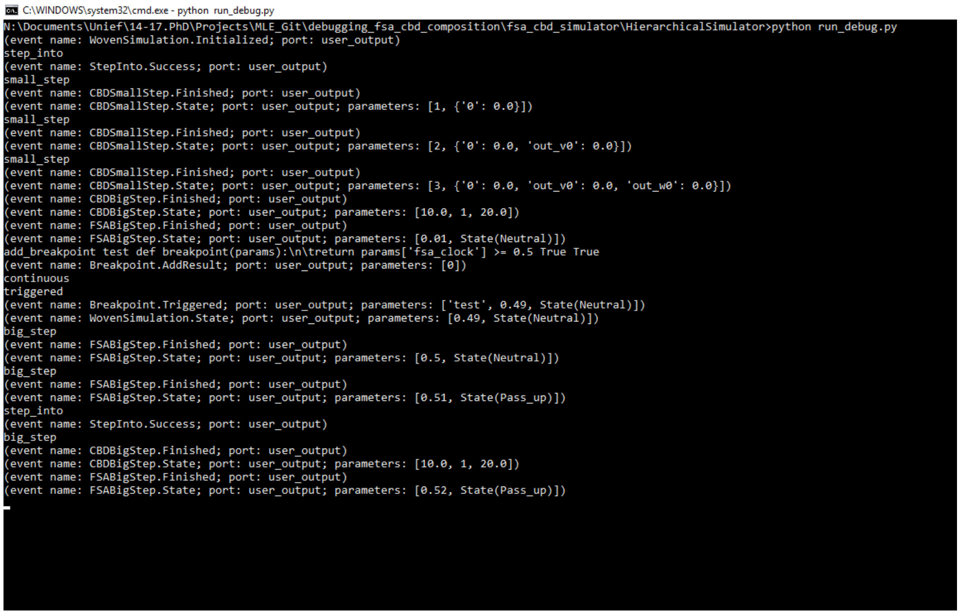
Figure 5.31: The de- and reconstructed T-FSA-CBD simulator.

simulator: instead of having one *Simulator* class, we have four: one for each level. The top-level simulator creates a new big step simulator and requests it to compute the next iteration of the simulation until the simulation end condition is satisfied. The big step simulator in turn creates a new small step simulator and requests it to compute the next state until the big step end condition is satisfied (*i.e.*, no more transitions are enabled). The small step simulator creates the lowest level simulator, which is simply a function call that executes a transition. Each level finalizes and communicates its results to the level above when its end condition is satisfied. The user can start the simulation (by instantiating the top-level simulator) and wait until it finishes execution, and inspect its results.

Analogously, a hierarchical canonical version for the CBD simulator can be constructed and instrumented with debugging support. By enhancing the simulation algorithm(s) with debugging support, their interfaces change, giving more control to the user to interrupt and control the simulation algorithm with debugging operations, outlined in Section 5.7.2. This interface is used to create a debugging-enhanced hybrid simulator.

In Figure 5.31, the hierarchical composition of the hybrid simulator is shown. The protocol of the simulation algorithm, as well as boundary concepts (such as state event detection) is implemented by the hybrid simulator. It has two child simulators, whose behaviour it controls through their exposed debugging interfaces. The *continuous* and *realtime* modes of the child simulators are no longer used, since that would not allow the hybrid simulator to do state event detection. Instead, the hybrid simulator controls its child simulators by letting them perform individual (small or big) steps.

The hybrid simulation algorithm can be fit into the generic template of Figure 5.29. The concept of ‘big step’, ‘small step’, state, and time operations remain unchanged from the ‘master’ T-FSA algorithm, whose semantics were used as a starting point for the Hybrid Automata simulator. The Hybrid Automata simulator processes states containing a CBD model as normal T-FSA states, and it controls how simulated time advances. The hybrid simulator is responsible for invoking the CBD simulators for states containing a CBD state.



```
C:\WINDOWS\system32\cmd.exe - python  run_debug.py
N:\Documents\Unief\14-17.PhD\Projects\MLE_Git\debugging_fsa_cbd_composition\fsa_cbd_simulator\HierarchicalSimulator>python run_debug.py
(event name: WovenSimulation.Initialized; port: user_output)
step_into
(event name: StepInto.Success; port: user_output)
small_step
(event name: CBDSmallStep.Finished; port: user_output)
(event name: CBDSmallStep.State; port: user_output; parameters: [1, {'0': 0.0}])
small_step
(event name: CBDSmallStep.Finished; port: user_output)
(event name: CBDSmallStep.State; port: user_output; parameters: [2, {'0': 0.0, 'out_v0': 0.0}])
small_step
(event name: CBDSmallStep.Finished; port: user_output)
(event name: CBDSmallStep.State; port: user_output; parameters: [3, {'0': 0.0, 'out_v0': 0.0, 'out_w0': 0.0}])
(event name: CBDBigStep.Finished; port: user_output)
(event name: CBDBigStep.State; port: user_output; parameters: [10.0, 1, 20.0])
(event name: FSABigStep.Finished; port: user_output)
(event name: FSABigStep.State; port: user_output; parameters: [0.01, State(Neutral)])
add_breakpoint test def breakpoint(params):\n\treturn params['fsa_clock'] >= 0.5 True True
(event name: Breakpoint.AddResult; port: user_output; parameters: [0])
continuous
triggered
(event name: Breakpoint.Triggered; port: user_output; parameters: ['test', 0.49, State(Neutral)])
(event name: WovenSimulation.State; port: user_output; parameters: [0.49, State(Neutral)])
big_step
(event name: FSABigStep.Finished; port: user_output)
(event name: FSABigStep.State; port: user_output; parameters: [0.5, State(Neutral)])
big_step
(event name: FSABigStep.Finished; port: user_output)
(event name: FSABigStep.State; port: user_output; parameters: [0.51, State(Pass_up)])
step_into
(event name: StepInto.Success; port: user_output)
big_step
(event name: CBDBigStep.Finished; port: user_output)
(event name: CBDBigStep.State; port: user_output; parameters: [10.0, 1, 20.0])
(event name: FSABigStep.Finished; port: user_output)
(event name: FSABigStep.State; port: user_output; parameters: [0.52, State(Pass_up)])
```

Figure 5.32: The user interface for the Hybrid Automata language.

God events can enable a transition at the T-FSA level.

Certain debugging operations are also valid at the CBD level, however. A user debugging a hybrid simulation might want to step through CBD block computations or change a signal value through a (CBD) god event. Because of the hierarchical nature of the formalism (CBD models are contained in T-FSA states), we define a *step into* debugging operation. This operation is valid when the simulation is paused, and the current T-FSA state contains a CBD model. It switches the execution context to the CBD simulator and enables regular debugging interaction at that level. The user can then execute a big step, a small step, or a god event. When a big step finishes, control is returned to the hybrid simulator.

Summary We demonstrated in this subsection that the debugging operations described in the previous subsection have been successfully added to the hybrid T-FSA-CBD simulator. In the next subsection, we couple the instrumented simulator to a debugging environment.

5.7.4 Debugging Environment

The debugging environment is a console application, shown in Figure 5.32. It is a proof-of-concept, which demonstrates debugging for hybrid formalisms is possible using our modified de- and reconstruction approach. A visual debugging environment can be developed, and would be a combination of a FSA debugging environment (similar to the Statecharts debugging environment presented in Section 5.4) and a CBD debugging environment (see Section 5.2).

At the start, the simulation is initialized to the *Started* state. This is a *CBDState* instance, and a *step_into* command allows the user to step through individual block computations. After three *small_step*s, the values of all blocks in the state are computed: a CBD big step ends, and with that, a T-FSA big step as well, bringing the system to the *Neutral* state. The user then adds a breakpoint, that automatically pauses the simulation when the clock of the T-FSA simulator has a value larger than 0.5. The simulation is then run in *continuous* mode, and the breakpoint triggers at time 0.49. This is expected, as in realtime mode, the time that is checked is the next transition time in T-FSA, not the current time. So, the breakpoint pauses the simulation one big step "early". Two big steps later, the simulation is in the *Pass_up* state. The user steps into this state, since it contains a CBD model, and executes a *big step*. The step computes all values of the blocks.

Summary

This section demonstrates how debugging support can be implemented for a hybrid formalism that combines T-FSA with Causal Block Diagrams. The language exhibits the following semantic features of the feature diagram presented in Figure 4.2:

- It is **hybrid** (through **embedding**).
- It allows to model **algebraic loops** (that need a specific schedule to be computed in a consistent manner).
- It implements **discrete-event** semantics, combined with **discrete-** or **continuous-time** semantics (depending on the types of blocks used in the instance models).
- It is **deterministic**.
- It has **static structure**.

Its hybrid nature is the most prominent feature of the formalism, as that requires the construction of a coordinator that manages the interaction between the two individual formalisms' simulators (using "glue" constructs). In this section, we have demonstrated that our approach can support debugging for languages exhibiting this feature.

5.8 A Domain-Specific Formalism for Production Systems

This section presents a domain-specific language for describing Armoured Personnel Carrier (APC) production lines (called the ProductionSystem DSL). This language differs from the previous languages for which we presented a debugger, which were general-purpose, whose semantics were operationally defined. In contrast, the ProductionSystem language's semantics is defined translationally, by mapping onto a formalism with known semantics. The ProductionSystem language is a causal, timed (discrete-event), static-structure, deterministic formalism.

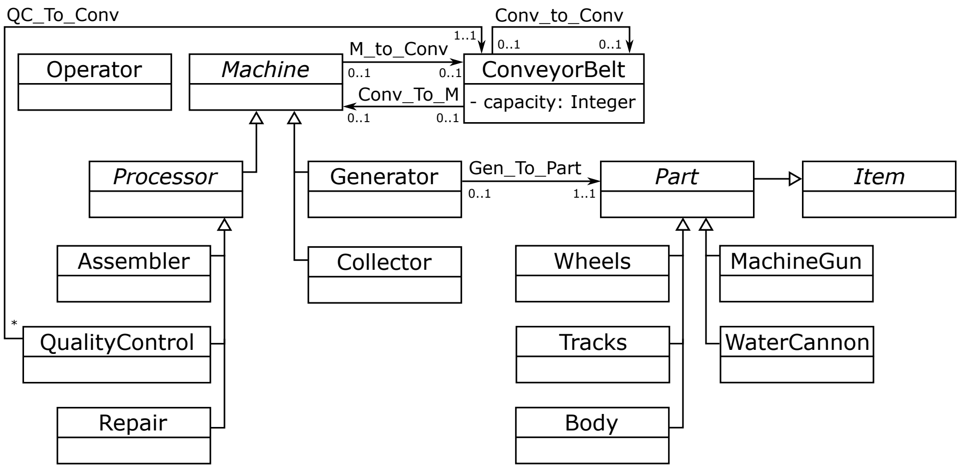


Figure 5.33: The metamodel of the ProductionSystem design language.

5.8.1 Syntax and Semantics

The ProductionSystem DSL has the following syntax requirements:

- A production system is a set of connected machines and conveyor belts.
- A conveyor belt carries (unfinished) products.
- All machines, except the collector, are connected to exactly one conveyor belt on which they can put parts and/or assembled products.
- Machines can either generate the parts of a APC (wheels, tracks, bodies, machine guns or water cannons), process parts and/or assembled products, or collect assembled products.
- Machines that process can be assemble machines, quality control machines or repair machines.
- Machines that process are connected to exactly one conveyor belt from which they take products that are to be processed.
- Machines that process can be operated by an operator.
- Conveyor belts can be connected.
- Quality control machines are a special kind of machine, as they can put products that need to be reworked on a different conveyor belt.

Figure 5.33 shows the metamodel implementing the requirements of this design language as a class diagram. The design language enables a modeller to build a production system consisting of machines (that either generate parts, process parts, or collect finished products), that are connected by conveyor belts. Operators are modelled as stand-alone entities; they are not connected to machines in the design language as this is a runtime concern only. The design language only allows to define a number of operators that are available at runtime to operate processors. Generators are connected to the part they generate, to avoid having to define many subclasses. Conveyor belts have a capacity: the number of items they can carry. Conveyor belts cannot have any items placed on them in the design language, as that is a runtime concern.

The ProductionSystem DSL has the following semantic requirements:

- There is only one direction in which products can be moved from one conveyor belt to the other.
- Products can move from a conveyor belt to a connected conveyor belt.
- Generators can put generated APC parts on their outgoing conveyor belt.
- Processors can only process when there is an operator at the machine.
- An operator can move from one machine to the other if the required (unfinished) part(s) is/are ready for that machine.
- An assemble machine with an operator can take two tracks, a body and a machine gun from its input conveyor belt and process them into a war APC that is put on the output conveyor belt, or can take four wheels, a body and a water cannon from its input conveyor belt and process them into a riot APC that is put on the output conveyor belt.
- A repair machine with an operator can take an (unfinished) product from its input conveyor belt, repair it, and put it on the output conveyor belt.
- A quality control machine with an operator can take an assembled item from its input conveyor belt. In case the item is not broken, it puts the item on its normal output conveyor belt. In case the item is broken, it puts the item on its other output conveyor belt for pieces that need to be reworked.

ALGORITHM 10: The simulation algorithm for the ProductionSystem DSL.

```
Input: Model to simulate ($model$)
$time \leftarrow 0$;
$state \leftarrow initialize\_state(model)$;
while not $endCondition(state, time)$ do
    while not $endConditionProductionSystemStep(state, time)$ do
        while not $endConditionMachineStep(state, time)$ do
            while not $endConditionBigStep(state, time)$ do
                $transitioning \leftarrow find\_transitioning\_elements(model, state, time)$;
                $output \leftarrow compute\_output(model, state, time, transitioning)$;
                $state \leftarrow compute\_new\_state(model, state, time, transitioning)$;
            end
            time $\leftarrow increment\_time(model, state, time)$
        end
    end
end
```

Now, an operational definition of the simulation algorithm for the ProductionSystem language has to be provided. However the simulator is implemented (operationally or translationally), we need a specification of its intended behaviour, which will later be used to define useful debugging operations. The simulator has the responsibility of initializing the runtime state and time, simulate the system until an end condition specified by the user has been satisfied, and return the resulting state at the end of simulation. The specification of the simulation algorithm is shown in Algorithm 10. During simulation, the state of the system is updated according to the required semantics defined above. The smallest steps of the simulation algorithm performs the following actions:

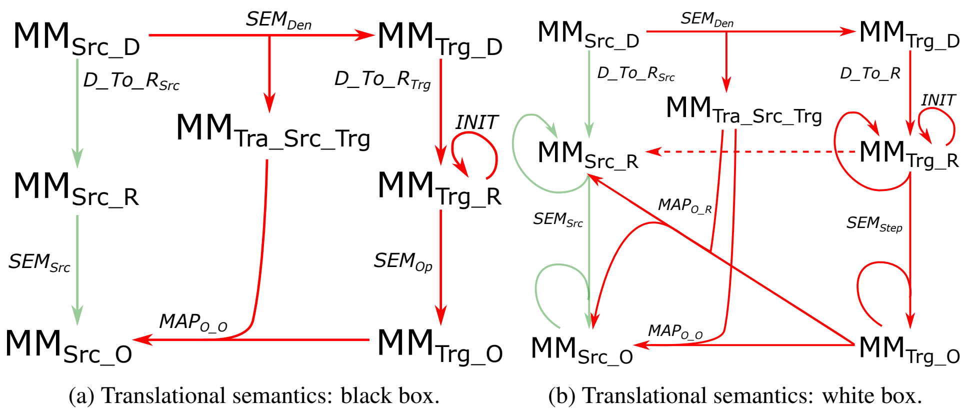
1. Find all elements that will perform an action. This can include:
    - Generators that are ready to generate an item. We assume a certain (randomized) delay between items being generated.
    - Items that can be moved from a conveyor belt to the next conveyor belt or a machine. We assume a fixed delay for an item to remain on a conveyor belt (proportional to the conveyor belt's capacity).
    - Machines that are ready to process, for which a (free) operator has to be found.
    - Machines that are ready to process and have an operator operating them; we assume a (randomized) delay for each machine to perform its task.
2. Compute the output of each element that performs an action. This can be, for example, an item that moves from one conveyor belt to the next.
3. Compute the new state of all elements that perform a transition.

We further define the following steps:

- A *big step* is a succession of one or more small step(s). After a big step ends, the time is incremented.
- A *machine step* is a succession of one or more big steps, and ends when one of the following is true:
    - A processor is ready, meaning it has received the necessary item(s).
    - A processor starts processing, meaning an operator was found to operate the processor.
    - A processor ends processing, meaning the item it worked on is placed on its output conveyor belt.
- A *production system step* is a succession of one or more machine steps, and ends when a new finished product is received by a collector.

This algorithm can be implemented operationally in a suitable language, but we choose to translationally map ProductionSystem models to the discrete-event formalism Parallel DEVS (for which a debugger was presented in Section 5.3).

Figure 5.34 illustrates how translational semantics work. We start by explaining the black-box approach presented in Figure 5.34a. We assume a target language with known semantics, whose design language, runtime language, and output language are described by the metamodels MM_{Trg_D}, MM_{Trg_R}, and MM_{Trg_O}. Four operations are defined:



(a) Translational semantics: black box. (b) Translational semantics: white box.

Figure 5.34: Translational semantics.

1. SEM_{Den} transforms valid models in the source design language described by MM_{Src_D} to valid models in the target design language described by MM_{Trg_D}. This produces a traceability model conforming to $MM_{Tra\_Src\_Trg}$ that keeps track of the mapping information between the source and target model.

2. D_To_R transforms valid design models in the target language to valid runtime models. This is most often a retype operation, since the runtime model is an extension of the design model. In some cases, the operation can be more involved, including exposing structural runtime information of design elements, such as in the Parallel DEVS debugger (see Section 5.3).

3. $INIT$ initializes the runtime model: this is the first step in the operational semantics.

4. SEM_{Op} is a black-box transformation which produces, from an initial system state, a state trace of the system behaviour in the output language MM_{Trg_O}.

5. MAP_{O_O} maps the output trace of the target language back to an output trace in the source language. It uses the forward mapping information stored in a model conforming to $MM_{Tra\_Src\_Trg}$ to back-translate the model. The result is a trace is a domain-specific simulation trace, using a technique similar to back-annotation, implemented by Hegedüs et al. [79].

Translational semantics allow to reuse existing infrastructure to simulate models. If done correctly, however, a user of the source language needs not know that the semantics are defined translationally: from the outside, a request for simulating a model in the source language results in an output model just the same as it would do if the semantics had been defined operationally. Instead of defining SEM_{Op_Src} operationally, we can deduce from Figure 5.34a the following equation:

$$SEM_{Op\_Src}(M_{Src}) = (MAP_{O\_O} \circ SEM_{Op} \circ INIT \circ D\_To\_R \circ SEM_{DEN})(M_{Src})$$

In our discussion, we have omitted intermediate runtime states of the source model and assume the user is only interested in the output model. For debugging purposes, however, we need to be able to inspect intermediate states. This is facilitated by defining two more operations:

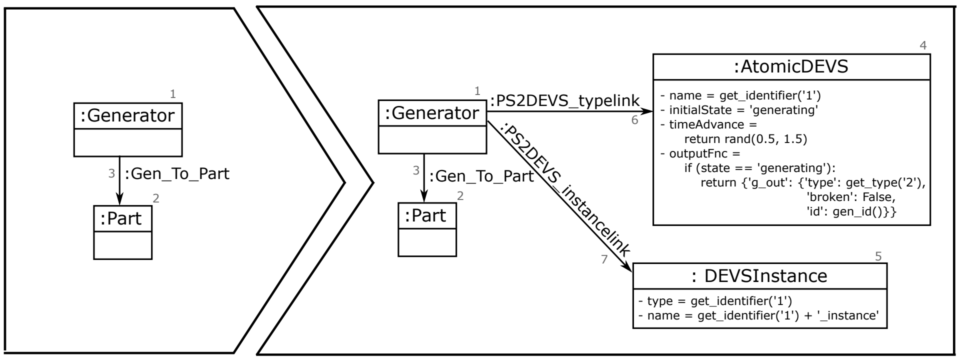


Figure 5.35: A rule that maps a generator onto its corresponding Parallel DEVS structure.

1. SEM_{Step} breaks open the black-box of SEM_{Op} by exposing the iterations of the simulation algorithm as *steps*. The output trace conforming to MM_{Trg_O} is no longer constructed in one operation. Instead, it is constructed iteratively by taking the current state (an instance of MM_{Trg_R}) and the current (partial) trace (an instance of MM_{Trg_O}), and producing the new state (an instance of MM_{Trg_R}) and the new (partial) trace (an instance of MM_{Trg_O}).

2. Map_{O_R} takes a (partial) trace (an instance of MM_{Trg_O}) and the traceability information encoded in an instance of $MM_{Tra\_Src\_Trg}$ and produces a new runtime state in the source language (an instance of MM_{Trg_O}), as well as a new trace model in the source language (an instance of MM_{Src_O}). These functions allow to construct a domain-specific trace iteratively based on the semantics of the target language.

To implement the semantics of the ProductionSystem DSL, we choose to map to the *Parallel DEVS* formalism. This is a natural choice, as 1) the users of the ProductionSystem DSL are mainly interested in throughput information (for which the Parallel DEVS formalism is well-suited), and 2) the semantics of Parallel DEVS are closely related, but sufficiently general to serve as a semantic domain. The translational mapping needs to be a total function. We explain below how the mapping is implemented:

- All model elements are mapped to (one or more) *DEVSInstance* instances. The type of these instances, an *AtomicDEVSBlock*, is either generated once for those elements whose behaviour is identical, or specific to this instance.

- A model element is linked to its type (an *AtomicDEVSBlock*) and its instance (a *DEVSInstance*) through traceability links.

- Two *AtomicDEVSBlock* instances are defined statically for operators and for conveyor belts of capacity 1. This means all operators and all conveyor belts share their behaviour.

  A conveyor belt in Parallel DEVS holds an item it receives through its input port for 1 time unit. It then sends it to the next connected instance.

  An operator in Parallel DEVS starts by waiting for a message on its input ports. If it receives a "query" message, it replies stating whether it is currently free to work or not. If it receives a "request" message, a machine requests the operator to work at that machine. In case the operator is free when the message arrives, the operator is allocated to that machine. The operator sends a message to the machine to acknowledge that the request was honoured. In case the operator is not free, he queues the request and waits until he is free again to honour the request. An operator is notified by the machine if his job has finished; the operator receives a "release" message.

- Operators are mapped to a *DEVSInstance* with type the *AtomicDEVSBlock* defining the behaviour of operators.
- Conveyor belts are mapped to a number of connected *DEVSInstances* equal to their capacity. The type of these instances is the *AtomicDEVSBlock* defining the behaviour of conveyor belts. This is a 1-n mapping: one instance in the DSL is represented by multiple instances in the semantic domain.
- Each generator is mapped to a *AtomicDEVSBlock* generated specifically for them: the generated type knows which type of item to generate, and does this after a randomized period of time has passed. The generator is also mapped to a *DEVSInstance* of the generated type. This instance is connected to the (first) *DEVSInstance* of its outgoing conveyor belt.
- Each collector is similarly mapped to an *AtomicDEVSBlock* and a *DEVSInstance* of that type. The collector receives finished products on its input port and increments its corresponding state variable(s) when that happens.
- Each processor is similarly mapped to an *AtomicDEVSBlock* and a *DEVSInstance* of that type. A processor can be "ready": for assemblers, this means the necessary parts have arrived to start producing an APC; for quality control machines and repair machines, this means a finished product has arrived on its input port. When a processor is ready, it searches for a free operator by sending messages to the operators according to the protocol previously described. Once an operator is found, the processor can start "working". The processor is completes processing after a randomized time interval. Assemblers put the finished product (which, with a random chance, can be broken) on their output port. Quality control machines put the finished product on the correct output port, depending on whether the product was broken or not. Repair machines put the repaired product on their output port.

These rules are encoded in a model transformation. An example rule is shown in Figure 5.35. It shows how a generator is transformed to its type and instance in Parallel DEVS. To be able to identify the type of each generator, the identifier of the generator (which is unique in the model being transformed) is taken as type name. The instance has the same name, with the string '_instance' appended.

5.8.2 Debugging Operations

We define the following debugging operations:

- **[Continuous Simulation]** In this mode, the simulation runs as fast as the underlying hardware can manage and the operating system will allow, until the end condition is satisfied. It is comparable to running program code, which is always run as fast as possible. At the end, the user can inspect the generated trace and any metrics collected.

- **[Real-Time Simulation]** In this mode, simulated time is synchronized with the wall-clock time. For debugging purposes, a scale factor can be applied to speed up or slow down simulation, while retaining the *linear* relation between simulated time and wall-clock time. A scale factor of 1 corresponds to real-time, while a scale factor smaller or greater than 1 slows down or speeds up simulation proportionally. State is observed throughout real-time simulation after each iteration of the simulation loop.

- **[Pause]** Pausing a simulation allows to inspect the current state of the system and enables other debugging operations, such as stepping and state modifications.

- **[ProductionSystem Step, Machine Step, Big Step, Small Step]** To transpose the notion of stepping, we turn to the simulation algorithm shown in Algorithm 10. We define four types of steps, corresponding to the three *while*-loops and the low-level computation steps. A *ProductionSystem Step* executes one simulation step (corresponding to an iteration of the outer *while*-loop): it completes the creation of a finished product. A *Machine Step* executes the system until one of the machines in the production system changes its state (*ready*, *start processing*, *end processing*). A *Big Step* executes a number of small steps, which are phases in the simulation algorithm. There are three such phases: find all elements that will perform an action, compute the output of each element that performs an action, and compute the new state of all elements that perform a transition. After every phase, relevant information is communicated to the user.

- **[Breakpoints]** A user can specify a breakpoint in the form of a condition on the execution state of the ProductionSystem simulation (including simulated time and current (total) state information). A breakpoint is a function that returns *True* when the simulation should pause, *False* in all other cases. When a breakpoint triggers, the user is notified of the name of the triggered breakpoint, and the current state of the simulation.

5.8.3 Implementation

This section explains our approach to implement a debugger for the ProductionSystem DSL. We explained the abstract syntax of the language, the abstract simulation algorithm, the translational mapping to Parallel DEVS, and the desired debugging operations. To implement the debugger for ProductionSystem, we reuse the debugger defined for the Parallel DEVS formalism.

Figure 5.36 presents the workflow for our approach, which is similar to back-annotation. We illustrate with our running example, which maps a production system language onto Parallel DEVS.

First, a design in the domain-specific language is created. This design model is subsequently transformed to a semantic domain, in our case Parallel DEVS. To start simulating, the

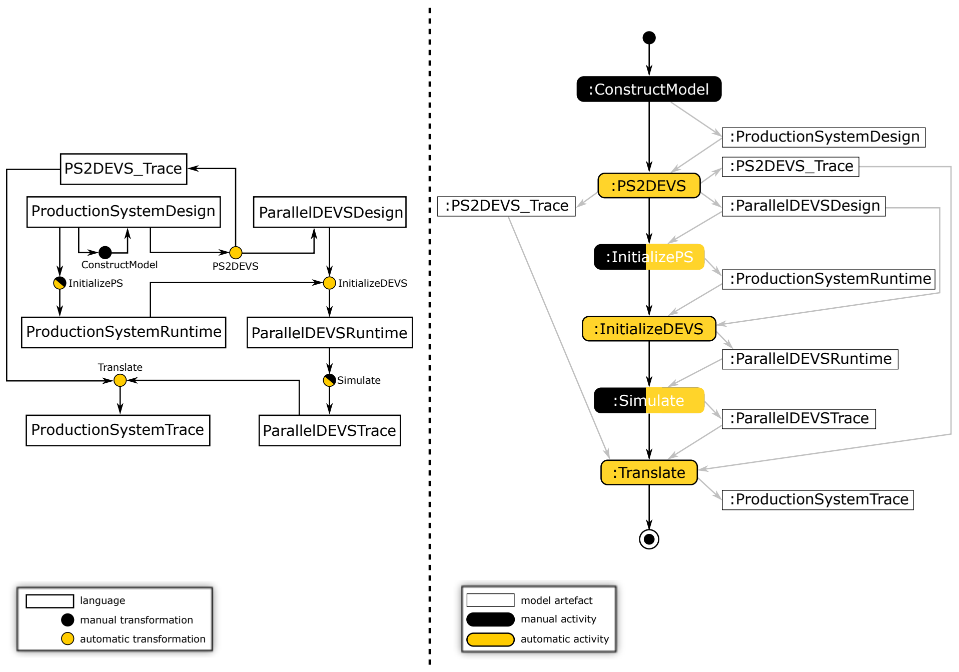


Figure 5.36: Workflow: translating a target output model back to the domain-specific level.

ProductionSystem model is initialized, resulting in the first snapshot of the domain-specific trace. The Parallel DEVS model also has to be initialized: since the initialization can depend on the initial state of the ProductionSystem model, a translation has to be made from the ProductionSystem state to the Parallel DEVS state. Once the models are initialized, the simulator starts simulating the Parallel DEVS model; this results in a trace. With back-annotation, the trace can be translated back to a ProductionSystem trace.

A debugging-enhanced simulator adds control and observation support to the simulation process. The trace no longer is communicated to the user at the end of simulation; instead, the trace is built incrementally and, depending on the operations the user performs, communicated at certain points in time. The translation between event traces is in that case no longer a model transformation that runs once. Instead, a debugger for the ProductionSystem provides similar levels of interactive support, meaning that the generation of the ProductionSystem trace also needs to be incremental. The debugger for the ProductionSystem language can reuse the debugging support offered by the Parallel DEVS debugger for both offering domain-specific debugging operations, as well as translating the (partial) trace to communicate to the user.

Two concepts are necessary to implement the desired debugging behaviour: event detectors and state propagators. These are explained in the next subsections.

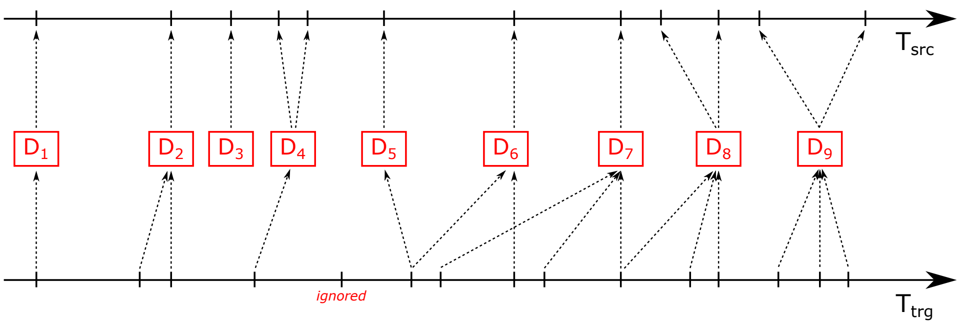


Figure 5.37: Detection: discrete-event.

Event Detection

Event detection is concerned with interpreting the events coming from the lower-level debugger. A series of such events can correspond to an event at the DSL level. These DSL events convey a state change of the DSL model. Detectors are defined for all high-level events of interest. The detectors are not responsible for translating the low-level state to a DSL state; that is the task of the propagators, explained in the next subsection.

Figure 5.37 shows graphically a trace of a discrete-event target formalism $\mathsf{F}_{\mathsf{trg}}$ with time base T_{trg}. Time flows from left to right, and at non-equidistant points in time, a state change is observed, either in the generated state trace, or while debugging the model. The goal is to detect patterns in this event trace and generate events that result in state changes in the state trace of the source language $\mathsf{F}_{\mathsf{src}}$ with time base T_{src} (which is translationally mapped onto $\mathsf{F}_{\mathsf{trg}}$). In the figure, we assume that T_{src} and T_{trg} are equal: they are both discrete-event.

We exhaustively discuss the possible relations between the two event traces: we explain using detectors D_1 through D_9. Some relations are not possible or not relevant for debugging. We explain why this is the case and which relations we do consider.

- D_1 is the trivial case: a state update at the target level corresponds to a state update at the source level.
- D_2 detects that one or more consecutive state updates at the target level correspond to a single state update at the source level. The state update at the source level immediately triggers once the last state update of interest at the target level is received.
- D_3 is an autonomous detector: it triggers without any event at the target level. Information is added to the source event trace which has no corresponding target events. This is not a useful detector, as we assume no information can be conjured without any relation to the target state trace.
- D_4 detects that one event at the target level corresponds to multiple events at the source level. This means that the debugger of the source formalism adds information in between state updates of the target level, information which is not available at the target level. This might be useful in certain cases. Assume that in our running

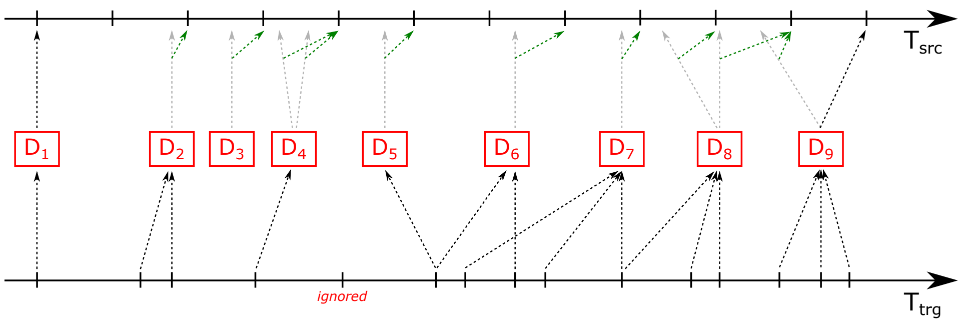


Figure 5.38: Detection: discrete-time.

example, we want to add animation: instead of an item moving from one conveyor belt to the next instantaneously (modelled by a departure and an arrival event that occur a non-zero amount of time apart), we show a fluent motion of the item (*i.e.*, a path interpolation between departure and arrival). This means the semantics were only partially implemented translationally: operational information is added at the source level. We do not consider such cases, as they would require an approach similar to hybrid debugging. If animation cannot be expressed in the target language, the target language should be changed, potentially to a combination of multiple languages.

- D_5 detects that an event on the target level corresponds to an event 'in the past' of the source level. Because of causality constraints, we consider such detectors invalid, as it is impossible to insert events in the past.
- D_6 and D_7 are two detectors that map multiple target events onto single source events, but those target events arrive out-of-order. No causality is violated, however, and such detectors can be useful.
- D_8 is an example of an m-n detector: in this case, 3 target events trigger 2 source events. The issue with this detectors is that they trigger when the last event is received. Otherwise, they should be modelled using one or multiple detectors such as D_2. Now, this detector schedules events in the past, leading to the same conclusion as for D_5.
- D_9 both schedules events in the past and in the future. Causality problems also occur with events in the future: assume another detector that affects the same element as the last event scheduled by D_9, but occurring before it. Consistency is not guaranteed.
- Events at the target level can also be ignored: this is useful in case the target level contains too much detail (transient states at the source level).

This discussion is also valid in case the detectors do not deal with state updates, but with other events coming from the target debugger. Examples include the triggering of breakpoints and small step information. From the discussion above, we conclude that $n - 1$ relations from target event traces to source event traces are allowed. These events can be interleaved, but no information can be added, meaning that no events can be autonomously scheduled. Scheduling events in the future or inserting events in the past is also not allowed.

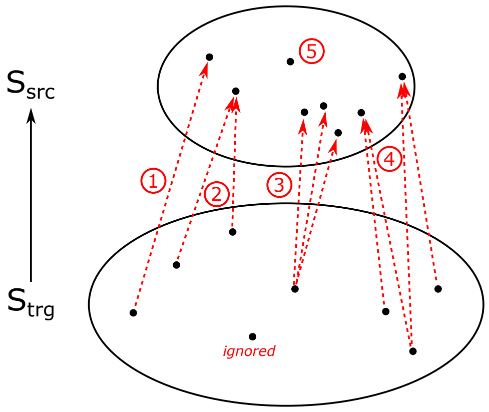


Figure 5.39: Propagation.

Figure 5.38 demonstrates how the time bases for the source and target formalisms can differ. The target formalism is still a discrete-event formalism, but the source formalism is discrete-time. The detectors remain the same, and so do our conclusions: D_3, D_4, D_5, D_8, and D_9 are not considered valid detectors. For the valid detectors, however, their detection often occurs 'in-between' the discrete time state updates. In that case, they are delayed until a discrete-time update occurs at the source level. This seems similar to scheduling events in the future, potentially violating causality if another detector schedules an event before that point is reached. In this case, however, *all* detectors schedule their events in the future. The relative order of simultaneous events is critical information, to ensure that the simulator can process them in order at the discrete state update times.

To implement detectors, they can be specified using temporal properties. These properties reason over a trace of events and *match* when the property is satisfied. A well-known language for expressing temporal properties is LTL, as implemented in model checkers such as SPIN [84]. More recently, ProMoBox [136] offers a domain-specific property language, allowing to define temporal properties using domain-specific syntax in the form of structural patterns that are connected using temporal operators. We model detectors using ProMoBox, after which the definitions are translated to a (query) model transformation that progressively matches the property while the trace is updated by the debugger. Currently, that translation is performed manually, but in the future we envision a fully automatic translation from ProMoBox properties to an operational form used to detect such events.

State Propagation

Detecting events at the source level is the first step towards translating the state trace from the target level to the source level. The next step is to process the information in the state update(s) and propagate that information from the target level to the source level.

Multiple relations between the target and source level states are possible. These are illustrated in Figure 5.39: the state set S_{trg} is mapped onto the state set S_{src}. The size of both sets was chosen deliberately: we assume the DSL is an abstraction whose state space is smaller compared to that of the target level. In some cases they might be equally large, but the source state space will never be larger.

We distinguish five possible state mappings, discussed next.

1. A *1-1* mapping, the basic case: one element in the state space of the target is mapped onto an element in the state space of the source.

2. An *n-1* mapping: multiple elements in the state space of the target are mapped onto one element in the source state space. This aggregation operation is relevant, since we assume the source is an abstraction.

3. A *1-n* mapping: one element in the state space of the target is mapped onto multiple elements in the source state space. This implies the source is a refinement of the target: information is necessarily added in the translation from target to source. But that information has to be stored somewhere. And if it is not in the target state space, it is not explicitly modelled but implicitly added in the state mapping. We do not consider such cases.

4. A *m-n* mapping: this is a generalization of the previous cases, and as long there is an abstraction relation between source and target states, our approach covers it.

5. A source state that is not the result of the translation of a target state. As was the case for the *1-n* mapping, information is retrieved from somewhere that is not the target state space. We do not consider such cases and assume all information can be retrieved from the target state space.

To implement propagators, a relation between elements in target states and elements in source states has to be defined, and operationally implemented to maintain that relation. One possibility consists of defining that relation declaratively, using, for example, triple graph grammars [179]. These definitions can be executed using an engine, ensuring that source and target states are constantly synchronized. Another possibility is to define propagators operationally using model transformation rules. Since propagation only needs to occur when a detector detects a domain-specific event, those are the points in time at which the state is propagated. Defining propagators as imperative transformation rules is therefore a natural choice.

A ProductionSystem Debugger

To define detectors and propagators that implement debugging support for the ProductionSystem DSL, we have to examine the interface provided by the Parallel DEVS debugger. To reiterate, the debugger accepts the following events:

- *continuous* runs the simulation as-fast-as-possible.
- *realtime* runs the simulation in real-time, optionally scaled.
- *pause* halts the simulation: either after the current big step ends, or as-fast-as-possible if, during realtime simulation, the simulator is in a waiting period.
- *big_step* executes one iteration of the simulation algorithm.
- *small_step* executes one phase of the current iteration.

- *add_breakpoint*, *del_breakpoint*, and *toggle_breakpoint* manages breakpoints: they can be added (which requires a name for the breakpoint, and a function which returns whether or not the breakpoint triggers based on the current state), they can be deleted, and they can be enabled or disabled.
- *inject* injects an event on a specified port.
- *trace* returns the full state trace.
- *reset* resets the simulation.

The debugger returns the following events:

- *terminate*: the simulation terminated.
- *breakpoint_triggered*: a breakpoint was triggered. Communicates the name of the breakpoint that triggered.
- *all_states*: returns the current state of all components in the model.
- *big_step_done*: a big step ended.
- *imminents*, *selected_imminent*, *outbag*, *inbags*, *transitioning*, *new_states*, *new_tn*: after each phase of a simulation iteration, an event is sent which contains the information that was computed during that phase.

The debugger for ProductionSystem is implemented by big stepping through the simulation of the Parallel DEVS model. After each big step, the information received from the Parallel DEVS debugger is passed to the detectors, and if one detects an event at the level of the DSL, the state information is propagated. We do not consider real-time simulation. This would require the simulator for ProductionSystem to have autonomous (operational) behaviour, implementing the waiting periods. We focus on the challenges arising from implementing a debugger for a formalism whose semantics are implemented translationally, and refer to our previous research for instrumenting operational semantics with real-time behaviour.

The following detectors are defined:

- A big step is detected when a big step in the Parallel DEVS simulator ends and one or more of the following conditions hold:
  - an item is received (from a machine or another conveyor belt) at the first *DEVSInstance* corresponding to a conveyor belt;
  - an item is received (from a machine or another conveyor belt) by the *DEVSInstance* corresponding to a processor or a collector;
  - an operator starts working at a processor, which is denoted by the state of the *DEVSInstance* corresponding to the processor storing the identifier of the operator in its state;
  - the state of the *DEVSInstance* corresponding to a processor changes to *waiting*, *ready*, or *working*.

- A machine step is detected when a big step in the Parallel DEVS simulator ends and the state of the *DEVSInstance* corresponding to a processor changes to *waiting*, *ready*, or *working*.

- A production system step is detected when an item is received by the *DEVSInstance* corresponding to a collector from its incoming conveyor belt.

- The end condition of simulation is detected when a big step in the Parallel DEVS simulator ends and the end condition (on the current ProductionSystem state) is satisfied.

Multiple detectors can detect an event: if a machine step is detected, so is a big step. But, if the user is executing a machine step, all big steps are ignored until a machine step is detected. If multiple conditions are satisfied for a big step (for example, two conveyor belts each receive an item in the same big step), these steps are aggregated: only one big step is detected at the DSL level.

Figure 5.40 shows one detector and associated propagator: the moving of an item on and between conveyor belts. The detector is defined as follows: *if a DEVSInstance, connected through a traceability link with a ConveyorBelt, that has no incoming connections from another DEVSInstance that is connected to that same ConveyorBelt has a non-null value for its 'item' state variable after that state variable has been null, a new item is detected.* In our example, a conveyor belt is defined with capacity 3, and is mapped to 3 *DEVSInstances* that can each hold one item. The first big step event coming from the target debugger has an item arriving in the first *DEVSInstance*. Since this item is new (it is not connected yet to the conveyor belt in the ProductionSystem model), the detector detects this as a big step. The propagator translates the runtime state of the target model: an item is created and connected to the conveyor belt. A next big step of the Parallel DEVS simulator moves the item from the first *DEVSInstance* to the second. The detector does not detect a DSL step: the event is ignored, but remembers that the *item* state variable of the first *DEVSInstance* is now null. In the last big step of the Parallel DEVS simulator, a new item is received by the first *DEVSInstance*. Again, a new item is created and connected to the conveyor belt. Using this detector and propagator, the state of a conveyor belt is an aggregation of the state of a number of elements in the Parallel DEVS model.

Figure 5.41 shows the detection of an operator starting to work on an assembler. There are five big steps involved, relating to the protocol which exists between a processor and operators:

1. A processor which is ready starts looking for an operator by sending a 'query' message to all operators.

2. The operators respond whether or not they are free.

3. The processor receives all responses and chooses an operator; the processor sends a 'request' message to that operator.

4. Eventually, the operator allocates itself to the processor.

5. The processor starts working.

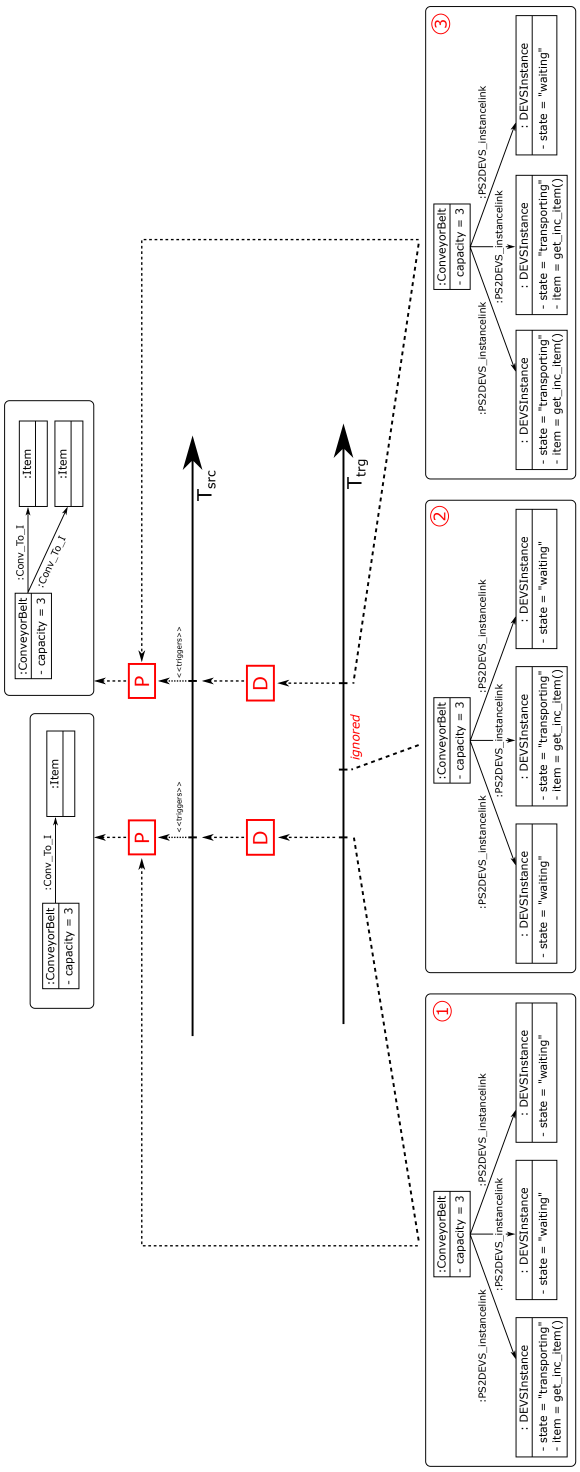


Figure 5.40: Detection and propagation: conveyor belt.

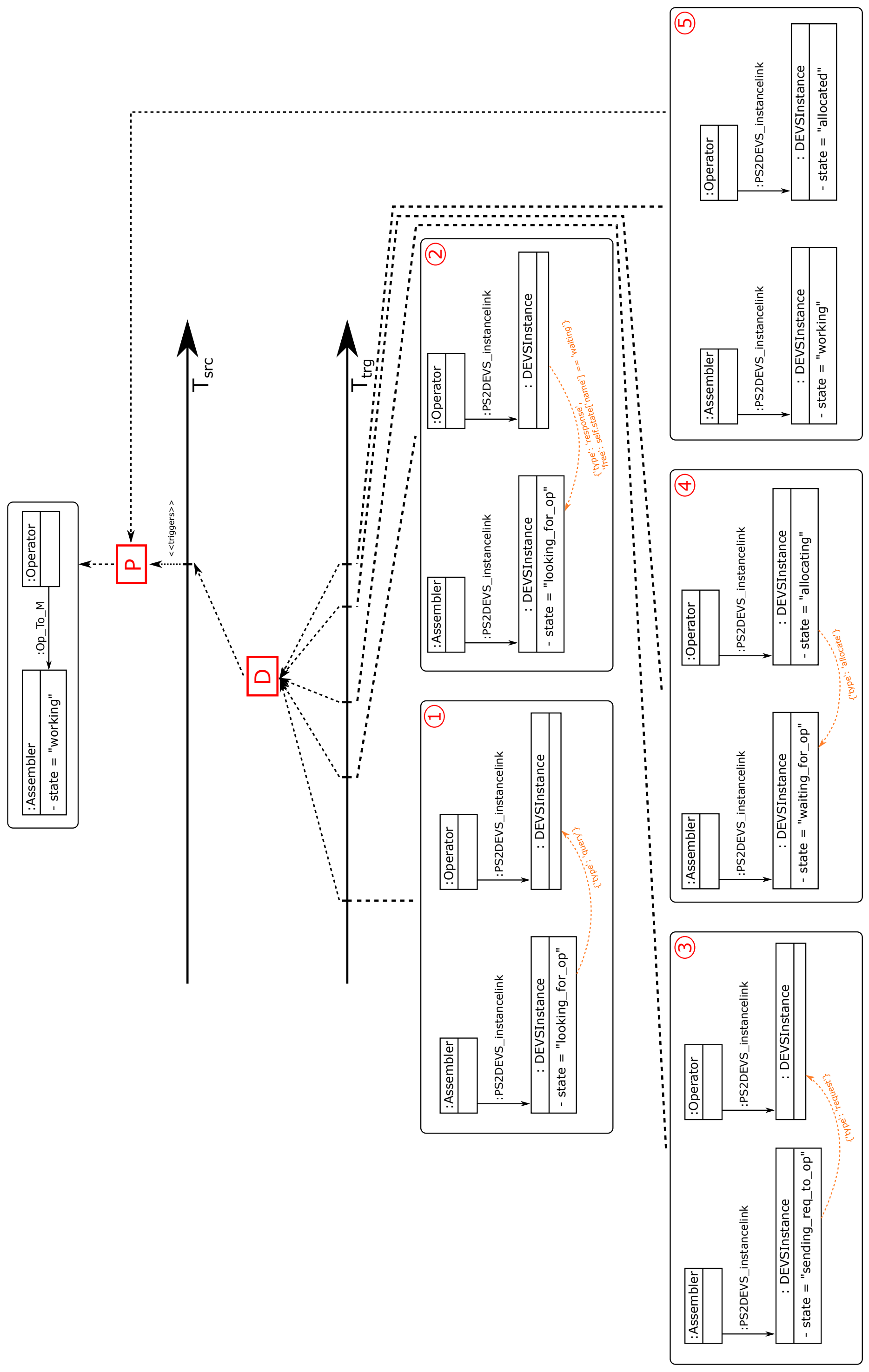


Figure 5.41: Detection and propagation: operator.

The propagation is *1-1*: the state of the Parallel DEVS elements representing the assembler and operator are mapped trivially to the DSL level. Detection, however, is *n-1*: multiple big steps are needed to detect that an operator has moved to a processor.

Implementing small steps is done similarly, but when a small step is requested by the user (at the ProductionSystem level), the system invokes Parallel DEVS small steps. We define three small steps, or phases, according to the ProductionSystem language's semantics:

1. Find all elements that will perform an action.
2. Compute the output of each element that performs an action. This can be, for example, an item that moves from one conveyor belt to the next.
3. Show the new state of all elements that perform a transition.

These small steps are implemented by detecting small step events coming from the Parallel DEVS debugger, in particular the following phases:

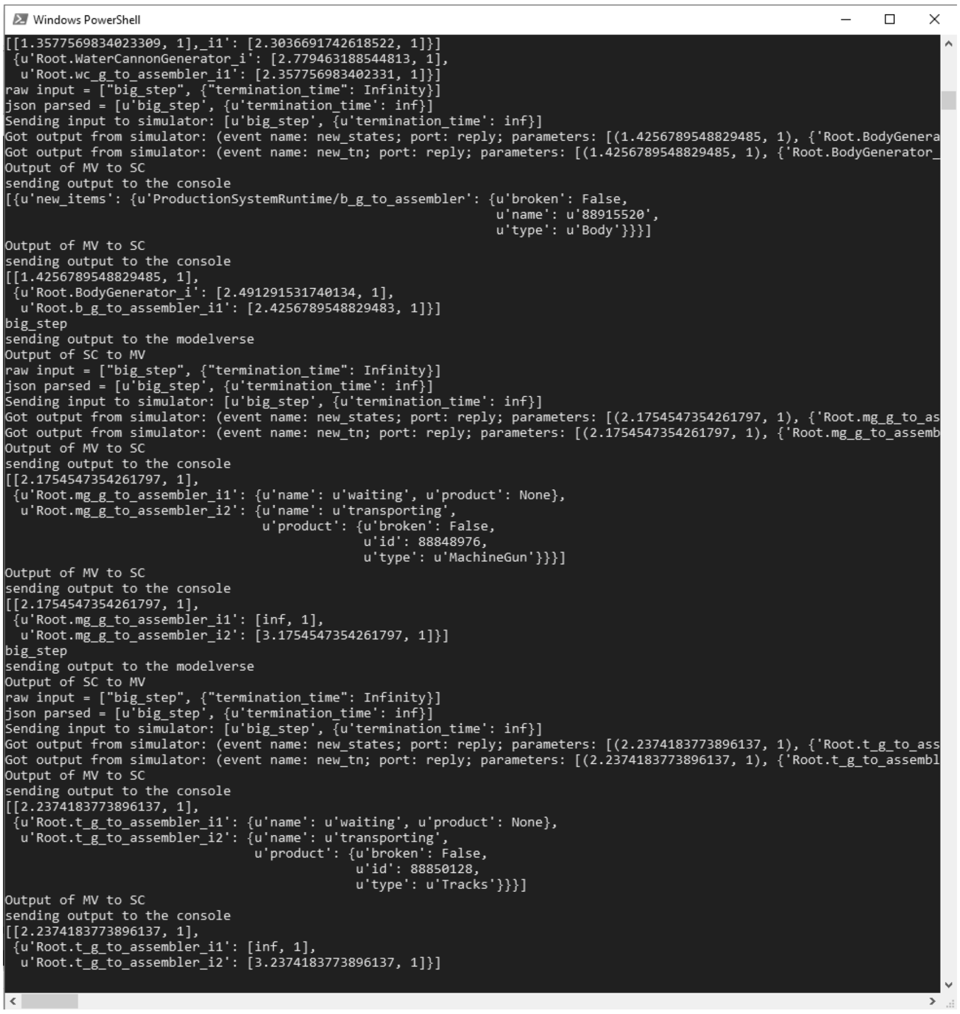
1. The *transitioning* message lists all components that will transition in the next step and can be translated, through the traceability links, to elements in the ProductionSystem model that will perform an action.
2. The *outbags* message lists all output generated by transitioning components. This output can be translated to items being transported between elements in the ProductionSystem model.
3. The *new_states* and *new_tn* messages are used to update the state of the components in the ProductionSystem model.

With detection and propagation, we build a debugger for the ProductionSystem DSL, while only relying on the traceability information that was a side effect of the translational mapping, and the debugging interface provided by the Parallel DEVS debugger.

Figure 5.42 presents the generated debugging environment for the ProductionSystem language. It is a textual console application, which allows a user to input debugging commands and displays the propagated state when a higher-level event is detected. In the example session, the user is "big stepping" through the simulation at the level of the production system. The states of the conveyor belts change as items are placed on them by generators, or passed from a connected conveyor belt. The generated interface demonstrates that it is possible to debug systems at the level of their operational semantics, even if the actual implementation uses translational semantics behind the screens. A more appropriate (graphical) interface can replace this proof-of-concept textual interface to debug ProductionSystem models using visualizations that are more familiar to the users of the language.

Summary

The goal of this chapter was to validate the de- and reconstruction approach for developing modelling language debuggers by applying the techniques presented in the previous chapter



Windows PowerShell

```
[[1.3577569834023309, 1],_i1': [2.3036691742618522, 1]}]
 {u'Root.WaterCannonGenerator_i': [2.779463188544813, 1],
  u'Root.wc_g_to_assembler_i1': [2.357756983402331, 1]}]
raw input = ["big_step", {"termination_time": Infinity}]
json parsed = [u'big_step', {u'termination_time': inf}]
Sending input to simulator: [u'big_step', {u'termination_time': inf}]
Got output from simulator: (event name: new_states; port: reply; parameters: [(1.4256789548829485, 1), {'Root.BodyGenera
Got output from simulator: (event name: new_tn; port: reply; parameters: [(1.4256789548829485, 1), {'Root.BodyGenerator_
Output of MV to SC
sending output to the console
[{u'new_items': {u'ProductionSystemRuntime/b_g_to_assembler': {u'broken': False,
                                                               u'name': u'88915520',
                                                               u'type': u'Body'}}}]

Output of MV to SC
sending output to the console
[[1.4256789548829485, 1],
 {u'Root.BodyGenerator_i': [2.491291531740134, 1],
  u'Root.b_g_to_assembler_i1': [2.4256789548829483, 1]}]
big_step
sending output to the modelverse
Output of SC to MV
raw input = ["big_step", {"termination_time": Infinity}]
json parsed = [u'big_step', {u'termination_time': inf}]
Sending input to simulator: [u'big_step', {u'termination_time': inf}]
Got output from simulator: (event name: new_states; port: reply; parameters: [(2.1754547354261797, 1), {'Root.mg_g_to_as
Got output from simulator: (event name: new_tn; port: reply; parameters: [(2.1754547354261797, 1), {'Root.mg_g_to_assemb
Output of MV to SC
sending output to the console
[[2.1754547354261797, 1],
 {u'Root.mg_g_to_assembler_i1': {u'name': u'waiting', u'product': None},
  u'Root.mg_g_to_assembler_i2': {u'name': u'transporting',
                                 u'product': {u'broken': False,
                                              u'id': 88848976,
                                              u'type': u'MachineGun'}}}]
Output of MV to SC
sending output to the console
[[2.1754547354261797, 1],
 {u'Root.mg_g_to_assembler_i1': [inf, 1],
  u'Root.mg_g_to_assembler_i2': [3.1754547354261797, 1]}]
big_step
sending output to the modelverse
Output of SC to MV
raw input = ["big_step", {"termination_time": Infinity}]
json parsed = [u'big_step', {u'termination_time': inf}]
Sending input to simulator: [u'big_step', {u'termination_time': inf}]
Got output from simulator: (event name: new_states; port: reply; parameters: [(2.2374183773896137, 1), {'Root.t_g_to_ass
Got output from simulator: (event name: new_tn; port: reply; parameters: [(2.2374183773896137, 1), {'Root.t_g_to_assembl
Output of MV to SC
sending output to the console
[[2.2374183773896137, 1],
 {u'Root.t_g_to_assembler_i1': {u'name': u'waiting', u'product': None},
  u'Root.t_g_to_assembler_i2': {u'name': u'transporting',
                                u'product': {u'broken': False,
                                             u'id': 88850128,
                                             u'type': u'Tracks'}}}]
Output of MV to SC
sending output to the console
[[2.2374183773896137, 1],
 {u'Root.t_g_to_assembler_i1': [inf, 1],
  u'Root.t_g_to_assembler_i2': [3.2374183773896137, 1]}]
```

Figure 5.42: The interface for the ProductionSystem debugger.

to a number of formalisms with diverse semantics. We created debuggers for the following formalisms:

- A procedural, sequential action language, to demonstrate the feasibility of our approach on traditional programming languages.
- Causal Block Diagrams, in which data flow can be modelled with blocks and signals. Blocks can form algebraic loops, which requires a (linear) solver to solve the set of equations they represent. This requires a particular ordering of block computations; this schedule needs to be presented by the debugger to the user.
- Parallel DEVS, a discrete-event formalism, which consists of components that communicate using events. A debugger for this language has to be able to show the communication patterns and the state changes of the components.

- Statecharts, a discrete-event formalism that is often used to describe the timed, reactive, autonomous behaviour of systems. It consists of states that can be composed hierarchically and orthogonally. Their semantics are intricate due to the interleaving of events raised and transitions executed in orthogonal components, as well as events coming from the environment.
- Petrinets, a formalism that is used to model non-deterministic behaviour. The semantics of Petrinets models branch: a debugger needs to be able to explore such branches.
- Dynamic Structure DEVS, an extension of Parallel DEVS in which structure-varying behaviour can be specified, often used to model multi-agent systems. A debugger for this formalism has to provide extra (visual) information to manage simulation entities being created and destroyed.
- Hybrid Automata, a hybrid formalism that exhibits both timed, reactive behaviour, as well as continuous behaviour. Since the two formalisms are hierarchically combined, a debugger needs to be able to switch contexts.
- A domain-specific language whose semantics are defined translationally, by mapping onto a formalism with known semantics (with an associated debugger). Since the simulator is not defined operationally, it cannot be instrumented as was the case for the other formalisms. Instead, a debugger needs to interpret state updates coming from the debugger for the target language.

By doing so, this chapter demonstrates the applicability of our approach by covering most of the language features in Section 4.1. We do not cover formalisms with spatial distribution semantics (such as Cellular Automata [214]), nor do we cover a-causal languages. We assume cellular automata are equivalent to discrete-event models (such as implemented by CellDEVS [210]), and as such, they do not present a new challenge for debugging. A-causal model debugging, on the other hand, has been covered by other researchers extensively. We refer the interested reader to the many references on this topic [26, 27, 157, 159, 186]). Additionally, while we show how to debug formalisms whose semantics are defined operationally and translationally, we do not consider the special case where the semantics are defined by code generation. Again, we refer to related work [47, 117, 216]. We also do not cover black-box simulators. From the debugger presented in Section 5.7, we can deduce that a grey-box approach is sufficient (white-box is not required) if the interface of the simulator exposes sufficient operations to manipulate the state of the simulation, and provides information on the current state of the simulation and its trace.

In the next chapter, we consider two advanced debugging techniques that are not implemented in the debuggers we presented up to now: omniscient debugging and live modelling.

Chapter 6

Advanced Techniques

In the previous two chapters, we presented a generic technique for constructing interactive debugging environments for simulation formalisms, which includes a workflow, architecture, and a structured approach to instrument an existing simulator with debugging support. We applied our technique to a number of representative formalisms to demonstrate feasibility. The debuggers all have support for a roughly equivalent set of debugging operations that can be categorized according to what they control: there are operations for observing and controlling *simulated time*, observing and modifying the *simulation state*, observing and controlling *simulation steps*, and *breakpoints*. This chapter explores two advanced debugging techniques: omniscient debugging and live modelling. Omniscient debugging adds a *step back* operation to a debugger, which allows users to step back in simulated time. Live modelling allows a user to change the design model of a running simulation. It merges the new design with the runtime state, such that from that point on, the simulation is run with the new model.

Structure Section 6.1 explains how omniscient debugging support can be added to a debugging-enhanced simulator. It acknowledges the overhead involved, and discusses a generic technique for selectively saving partial traces, to limit the memory consumption. Section 6.2 explains how live modelling can be implemented, by first deconstructing the live programming approach into necessary artefacts and operations, and transposing these to simulation languages.

6.1 Omniscient Model Debugging

In Section 5.3, we presented a debugger for Parallel DEVS. We now extend this debugger with a *step back* operation. Stepping back in time causes the simulation to revert to the state before the last big step. This is realized through state saving: every consecutive state is stored in memory. When requested, previous states are put in place again, and the global simulation state is reverted. This is done through serialization of the state, and subsequent deserialization when restoring the state. We call this technique *Full State Saving*: the complete model state is stored after each transition. Such a state saving technique is naive, however, and this section focuses on optimizing the implementation, to lower the threshold for the modeller to use it. All experiments in this section were performed on a desktop computer with an Intel i5-4570 (3.2 GHz) processor, 16GB of DDR3-1600 main memory and a 500GB 7200 rpm HDD.

6.1.1 Stepping Back

In contrast to the usual debugging operations, omniscient debugging goes back in time. This grants more freedom to modellers, as they can now traverse the simulation trace in two directions. As users are discovering the benefits, omniscient debugging is gaining popularity. Two ways of stepping back in time exist: taking a single simulation step back, or jumping to an arbitrary point in simulated time. We discuss stepping back in time in the context of the extended Parallel DEVS debugger. While *Full State Saving* is a relatively easy technique for implementing omniscient debugging, it is not efficient, both memory-wise and computation-wise, as at each step, the full state of the model is stored. A small, but significant, optimization can be made: only the state of atomic models that execute a transition is stored. We call this slightly optimized algorithm *Copy State Saving*. It doesn't store a single consistent state, but only timestamped partial states. These partial states can be combined into a single consistent global state. This decreases both time and memory consumption, as unchanged states are not serialized and stored again.

Despite omniscient debugging's advantages, there are severe performance limitations. First, omniscient debugging is plagued with memory issues [19, 43, 162]. Storing the complete simulation trace eventually leads to memory exhaustion, as trace size only increases. When simulators run out of memory, simulation halts. This makes it impossible to simulate large-scale models using omniscient debugging. Most omniscient debuggers tackle this problem in a lossy way. Either they use a time window, where only the most recent states are retained, or they only store part of the state. Both approaches are lossy: it is impossible to go beyond the fixed time window, or to access untracked parts of the model state.

Second, omniscient debuggers have low simulation performance [116, 161]. The primary overhead is in serializing and storing model states after each transition. Contrary to memory consumption, high time overhead does not prevent a model from being simulated. Nonetheless, it might become impossible to simulate the model within reasonable time bounds.

While the memory problem is obvious given the stored amount of data, the time problem is less obvious. For our Parallel DEVS debugger, Figure 6.1 shows the difference between turning omniscient debugging on and off, dependent on the size of the state. It shows an aggregation of 20 simulation runs of a benchmark Parallel DEVS model with a varying model state size. Without omniscient debugging, the model is simulated as-fast-as-possible. With omniscient debugging, the model is simulated as-fast-as-possible, but after each "big step" (state update), the current simulation state is saved to disk. We see that execution time increases as the state history increases, due to the serialization overhead of state saving becoming the bottleneck. This increase is linear, as the serialization routine used has linear complexity in terms of the state size. The variance observed throughout the runs was minimal, and is therefore not shown.

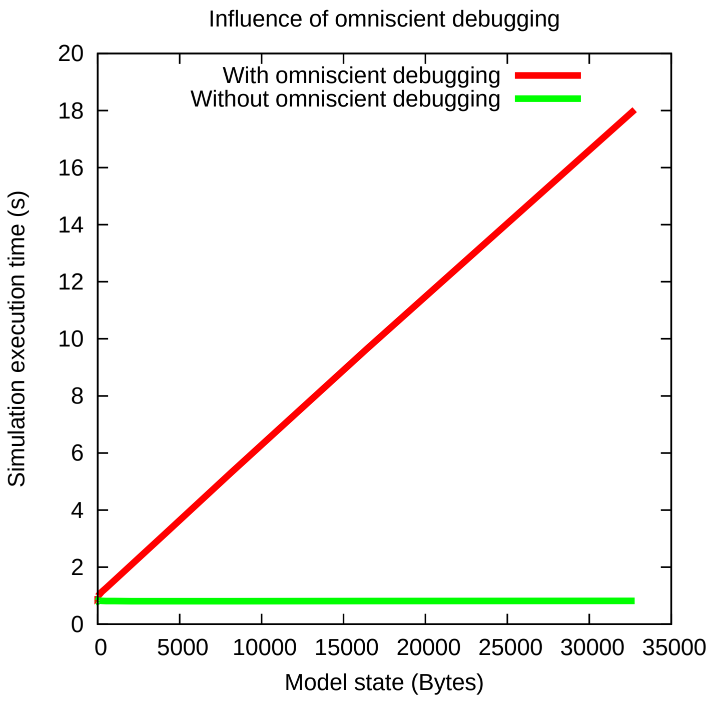


Figure 6.1: Overhead of omniscient debugging in forward simulation.

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 warrant the significant overhead on the more frequent forward simulation operations. The other option would be for users to select upfront whether they want to use omniscient debugging or not. This is not always known upfront, and even when users know so, simulation becomes too slow for practical use when enabled.

6.1.2 Optimization

We now optimize the state saving algorithm described previously. Before we do so, we go back to the core problem: “how to jump back to an arbitrary point in history?”. This is the same problem encountered in Time Warp [87], which we will now further investigate.

Relation to Time Warp

In parallel simulation, synchronization protocols are frequently used to allow different simulation nodes to be at different points in simulated time [58, 59], increasing parallelism. Time Warp is one such optimistic synchronization protocol. In Time Warp, each computational node simulates as fast as possible, ignoring the possibility for external events. If such an event occurs anyway, simulation is rolled back to a point right before the event was supposed to be processed. The event is then processed as usual, circumventing the causality problem. Rollbacks significantly resemble our core problem: a way to jump to arbitrary points in the past simulated time. We base our omniscient debugging algorithm on those defined for optimistic synchronization.

In the context of Time Warp, several algorithms were created [39, 165, 169], 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 size of the rollback (*i.e.*, the amount of steps that need to be rolled back) 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 [59]. 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. Whereas 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. Since periodic state saving's disadvantages are minimal for omniscient debugging, we use this algorithm for our implementation of omniscient debugging.

Periodic State Saving for Omniscient Debugging

Our algorithm 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, instead of simulated time. While simulated 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

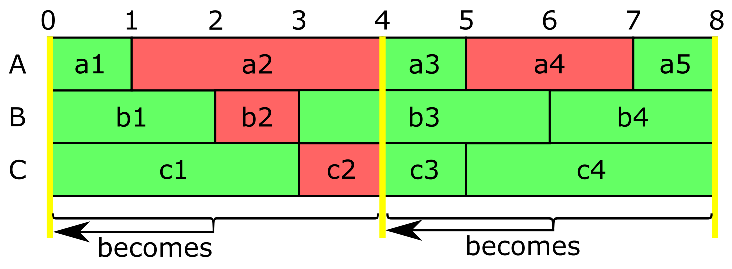


Figure 6.2: Overview of periodic state saving approach.
Green states (light) are stored, red (dark) states are not. Yellow lines indicate a point at which a snapshot is made.

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 simulated time.

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 noticeable performance impact.

We consider this approach and its configuration options in two dimensions: time and memory.

Time Before we talk about optimizing time, we reason about which operations to optimize: making one operation faster potentially makes other operations slower. In general, forward simulation steps are far more common than backward simulation steps. Forward steps happen at a high frequency, as they are automatically invoked. Backwards steps happen only rarely and are invoked interactively by the user. Whereas thousands of forward steps can be requested in a single second, backward steps occur only rarely: the modeller must analyse the model to decide whether to step back again or not, and must press a button to invoke the next step. Optimizing forward simulation should thus be our priority, on the condition that backwards steps remain responsive enough for interactive use.

It is clear from our approach that we prioritize forward simulation: the forward simulation algorithm is unaware of any omniscient debugging, and thus experiences no overhead. Sometimes, minor pauses are noticeable in the simulation, which are used to serialize the complete state. When rolling back, the latest snapshot is selected and simulation is restarted from there on. This is shown in Figure 6.2, where only three snapshots are made. Any rollback will be changed to a reset of a snapshotted state, and forward simulation continues from there on. 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.

Backwards steps, however, are clearly penalized, as they now require a *forward simulation phase*, in which the latest consistent snapshot must be simulated up to the requested point in 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.

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.

Memory We consider two types of memory: main memory and disk. While main memory is fast and easy to use, it is relatively small compared to the hard disk. Main memory size is therefore a problem, as all objects are stored in main memory by default.

Our approach tackles this by writing old state snapshots to disk, freeing up main memory. It is unlikely for old snapshots to be accessed, and since the additional delay is only in the order of tens of milliseconds, we can easily store old states to disk. Only a single snapshot is loaded from disk, as all snapshots are self-contained. Recent snapshots, having a higher chance of being accessed, stay in main memory. As we provide a lossless approach, we must keep these older states available for when a jump to them is requested, however small the chance of accessing them is.

Writing data to hard disk is possible for both copy state saving and periodic state saving. For copy state saving, the state of each model is managed individually, thus requiring as many disk accesses as there are atomic models. For periodic state saving, the full state at some point in time is managed as a single block of data, requiring a single disk access.

Although reading and writing data to disk does not seem very attractive from a performance view, performance is good due to asynchronous I/O. During forward simulation, where performance really matters, we only do sporadic writes to main memory. We only write to disk when main memory usage passes the defined threshold. These writes happen asynchronously and can therefore be considered (almost) instantaneous, as it will be buffered by the Operating System (OS). Reading data is the more expensive operation, but that only happens once per backward step. 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.

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. There are two directions to solve this problem.

The first possibility is to further extend storage media through existing technologies, such as adding more disks or storing it in the cloud. Whereas this technology exists and is sufficiently mature, this again merely delays the point where memory runs out: neither of these approaches has an infinite capacity, though it can be increased on-demand.

The second possibility truly tackles infinitely running simulations, 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, since 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, we can bound the time it takes to reach the requested state. This differs from a window-based approach: our approach is lossless. Our approach is also guaranteed to never be slower than restarting the simulation completely, as a restart always represents the worst case situation, in which there is no closer snapshot available.

6.1.3 Performance Evaluation

We now evaluate this algorithm in our Parallel DEVS debugger. We evaluate several kinds of resources, operations, and models. For resources, we measure CPU time needed for operations, main memory consumption, and disk space used. For operations, we consider the time needed for a forward step, a backward step, and a random jump to the past. For models, we define a minimal, synthetic benchmark model with a configurable state size. As the model is synthetic, we have disabled compression for these benchmarks, as it would skew the results: the data would either be trivially compressed (*e.g.*, all zero values), or not compressed at all (*e.g.*, full random).

First, we focus on the initial goal: minimizing time and space overhead of omniscient debugging. Second, we discuss trade-offs: omniscient debugging operations become slower. Finally, we vary the size of the benchmark model to measure the influence on performance.

For all dimensions, our benchmark model is the same simple coupled DEVS model with a configurable number of atomic models. Each of these atomic models has a configurable state size. The atomic models are configured to do an internal transition after a (uniformly distributed) time advance has passed. The structure of the coupled model (*i.e.*, how these atomic models are coupled) is irrelevant to the discussion and is therefore not discussed further.

6.1.3.1 Omniscient Debugging Overhead

The first advantage of our approach is decreased simulation overhead for forward simulation. In our problem statement, Figure 6.1 indicated the performance impact of omniscient debugging. Such overhead is unacceptable for complex and long-running simulations. Certainly since it is always imposed, even if no omniscient debugging features are actually used when debugging.

Our solution is periodic state saving, which significantly influences forward simulation speed, as shown in Figure 6.3. Figure 6.3 is an updated version of Figure 6.1, now including

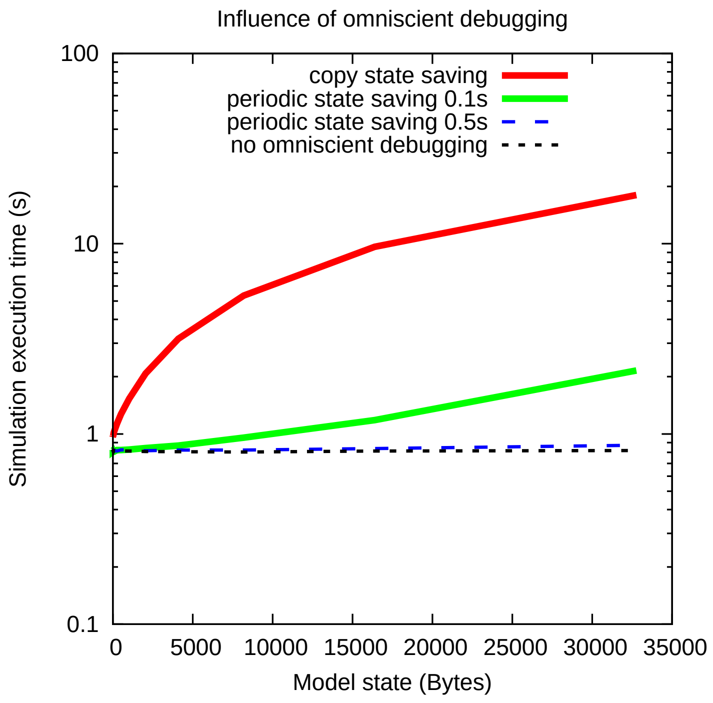


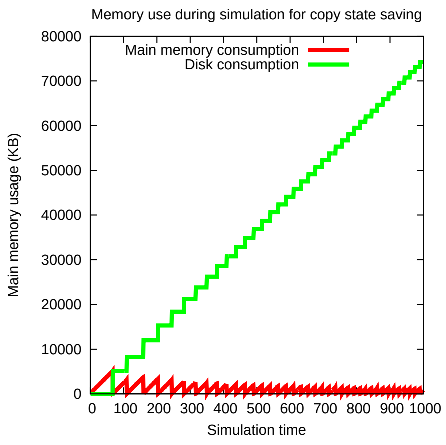
Figure 6.3: Overhead of omniscient debugging (logarithmic scale).

results for periodic state saving with two different configurations. Again, the graph shows an aggregation of 20 simulation runs of a benchmark Parallel DEVS model with a varying model state size. The variance observed throughout the runs was minimal, and is therefore not shown. Depending on the desired responsiveness of omniscient debugging operations, a latency of 0.5 seconds might be tolerable. In that case, forward simulation has barely any overhead, while a backwards step takes up to 0.5 seconds. Even with a latency of only 0.1 seconds forward simulation significantly outperforms copy state saving.

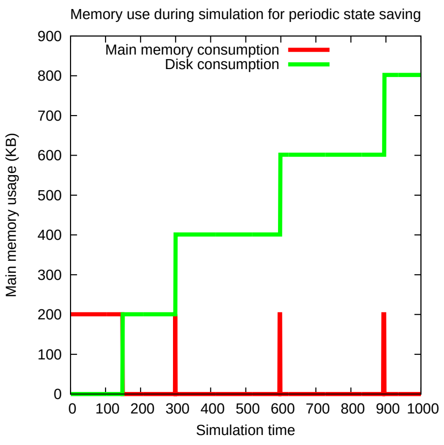
Memory and Disk Consumption

The prime concern with omniscient debugging is memory consumption, as the full state trace must remain accessible. But whereas copy state saving needs many different states to create a consistent snapshot, this is not needed with periodic state saving. Only one snapshot is stored every so often, which is guaranteed to be consistent. This drastically decreases memory consumption.

To demonstrate the difference between copy state saving and periodic state saving, we compare the performance results of both approaches in Figure 6.4. The results were obtained from a single simulation run, instead of an aggregate of multiple runs. We did this to clearly show, in the graph, the points in simulation time when the state is saved, illustrated by a “jump” in the plot. In an aggregate, these jumps would be smoothed, making it impossible to see the the points in time when a state save occurs.



(a) Memory use of copy state saving.



(b) Memory use of periodic state saving.

Figure 6.4: Memory usage of the two approaches.

The results for copy state saving are shown in Figure 6.4a, plotting the evolution of main memory and disk consumption. Main memory consumption increases linearly as new data is saved at each simulation step, and the cumulative number of simulation steps increases linearly due to the uniform time between transitions. After some time, data is moved from main memory to disk in big chunks. These points in time are clearly identifiable in the figure as the points where main memory consumption decreases and disk consumption increases. Note that, at simulated time 1000, approximately 70 megabytes of disk space is used to store all intermediate states. While this does not seem much, these figures are obtained after less than a minute of simulation on a simple model. For long running simulations with realistic models, disk space quickly runs out. Additionally, writing this much data to disk quickly becomes infeasible to be buffered by the OS.

Figure 6.4b shows results for exactly the same model, but with periodic state saving. Our first observation is that main memory remains constant most of the time, only increasing at points where a full snapshot is taken. Since a single copy of the simulation state is already too large according to our defined space constraint, the data is immediately swapped to disk afterwards. Another interesting observation is the total memory: a mere 800 kilobytes. Only four copies of the state are stored, being much less memory intensive than saving all intermediate states, even incrementally. As the time between two transitions is uniformly distributed, there is some (emergent) synchronization between the wall-clock time and simulated time, resulting in equidistantly spaced snapshots.

Jump Latency

Periodic state saving excels in memory consumption and simulation overhead. The drawback is slower backward steps: some forward simulation is necessary to get to the desired state.

The latency for jumps backward in time is shown in Figure 6.5a. Again, as in Figure 6.4, the results were obtained from a single simulation run, to clearly see how the latency "jumps". If an aggregate were plotted, the plots would again be smoothed, making it impossible to observe these jumps. For this benchmark, the model is simulated up to simulated time 100. From that point, we measure how long it takes to jump to a specific point in its state history. This point is seen on the x axis. Copy state saving has a near-constant delay, no matter to which point in simulated time is being jumped. This is to be expected, as each state is stored in memory and can just be retrieved. Cost of retrieval from memory, and even from disk, is negligible compared to the cost of forward simulation.

For periodic state saving, results are far worse: most points in simulated time require computation time for the forward simulation phase. This is to be expected: only some full states are stored in memory, and these will be the points to which a rollback occurs. From our results, these points seem to be somewhere around simulated time 0, 39, 75, and 89. Executing a rollback could, in the worst case, be to a point in time right before a snapshot is made (*e.g.*, time 38). We can also deduce from the results that our interval between two snapshots was 0.5 seconds, as a jump never takes longer than 0.5 seconds. Note the snapshot at time 89, which should not have occurred until the latency reaches almost 0.5s. This is probably caused by other functionality of the simulator (such as compressing and writing snapshots to persistent storage), which is taken into account in the interval, but doesn't contribute to simulation progression.

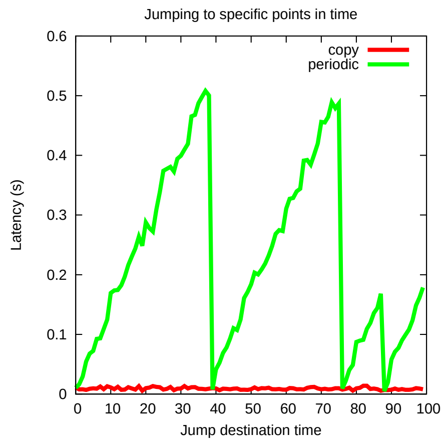
Influence of Model Size

Finally, we analyse the influence of the total size of the model on both state saving options, in terms of memory consumption. There are two dimensions influencing the size of the model: the number of atomic DEVS models, and the size of each model's state. Model state size was previously studied in Figure 6.3.

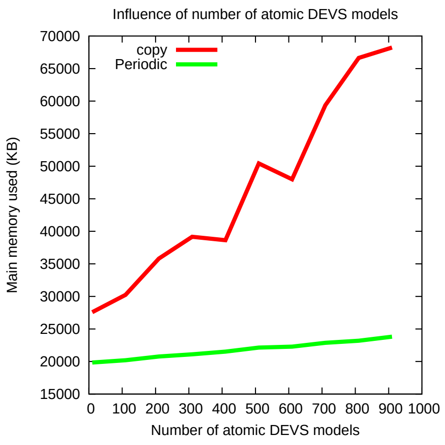
Figure 6.5b presents result for a varying number of atomic DEVS models. These results are an aggregate from 20 simulation runs. The variance observed throughout the runs was minimal, and is therefore not shown. Increasing the number of atomic DEVS models barely influences periodic state saving, as snapshots are only taken infrequently. Copy state saving is significantly impacted, as it stores, for n models, n history queues containing all previous states of the model. Increasing the number of atomic DEVS models also increases the number of executed transitions, requiring even more serialization and storage.

6.1.4 Conclusion

Despite the increasing popularity of omniscient debugging in the programming language domain, other domains are reluctant to incorporate it. For Parallel DEVS, our debugging environment is, to the best of our knowledge, the only simulation tool to implement it. Although fully implemented, performance was a major consideration, even if omniscient debugging is only used sparingly. Performance considerations exist in terms of memory and time efficiency. With naive algorithms, even small models become difficult to simulate due to its resource consumption. We therefore set out to find algorithms to cut down the overhead in terms of forward simulation performance and memory consumption.



(a) Jump latency for copy and periodic state saving.



(b) Influence of number of atomic models on memory consumption.

Figure 6.5: Comparing latency and the influence of model size.

We presented periodic state saving, combined with swapping to disk, as a lossless technique to restore state histories in a tolerable time. Forward simulation overhead was significantly reduced at the cost of slower omniscient debugging operations. Omniscient debugging operations do become slower, though they remain responsive for interactive use. Our synthetic benchmarks validated our expectations.

6.2 Live Modelling

In this section, we transpose the essence of live programming onto the modelling domain. 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 method presented in this section provides structure for language engineers who want to make their language live, which is currently still often considered a black art, even in the programming domain [28]. Our approach distils all aspects of liveness into a single operation, which we term *sanitization*.

Despite mostly being presented as a debugging operation in this section, live modelling can be applied to other situations as well, such as education. Providing support for “live modelling” was identified as a key feature to advance the usability of model-driven techniques [104].

This section is organized as follows. Section 6.2.1 presents an introduction to live programming, explaining the process in general. Section 6.2.2 introduces the necessary background on live programming and deconstructs it into its basic concepts. Section 6.2.3 transposes these concepts to live modelling, and presents reusable artefacts, operations, and a workflow for turning any modelling language live. Section 6.2.4 applies these techniques to two example languages.

6.2.1 Introduction to Live Programming

Live, or interactive, programming aims to bridge the "gulf of evaluation" [114, 200]. 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 their 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 [46], can be seen at `http://debug.elm-lang.org/`.

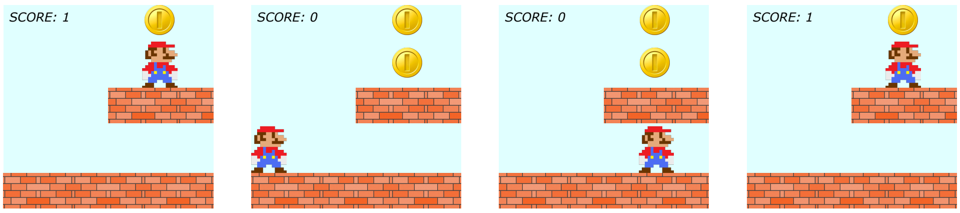
Basically, the process of live programming is as follows:

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 while it is running.
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 7^{th} 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). It 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, and many approaches exist. Three categories were proposed [130], 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 6.6a, 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. 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 re-initialized to zero. This is shown in



(a) Original configuration. (b) No live programming. (c) Recorded event. (d) Real-time.

Figure 6.6: State of the game before and after decreasing the jump height parameter.

Figure 6.6b, 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 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 [46]. 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 6.6c, 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 [64], 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 6.6d, 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.

6.2.2 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).

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. They consist 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, expressed in *ELF*. 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 it 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.

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

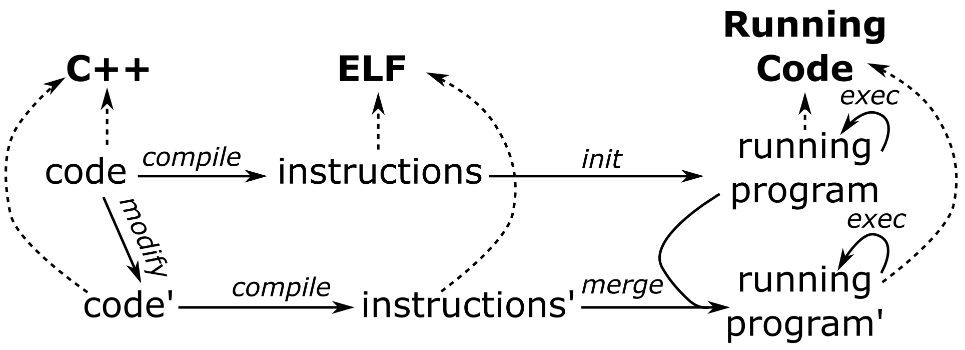


Figure 6.7: Diagrammatic overview of live programming.
Full lines represent operations, dotted lines represent typing relations.

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 6.7, 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.

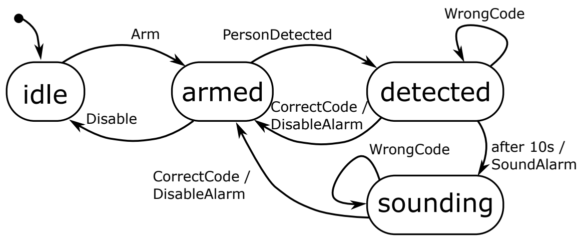
6.2.3 Transposing to Live Modelling

Taking the diagrammatic process presented in Figure 6.7, 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 metamodel. 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.

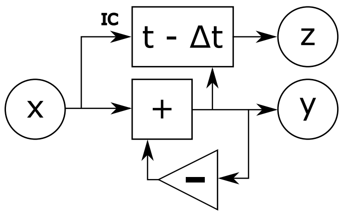
Artefacts

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 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. We show two example design models in different languages. An example T-FSA model



(a) Example FSA model of a home security alarm system.



(b) Example CBD model, containing an algebraic loop.

Figure 6.8: Example design models.

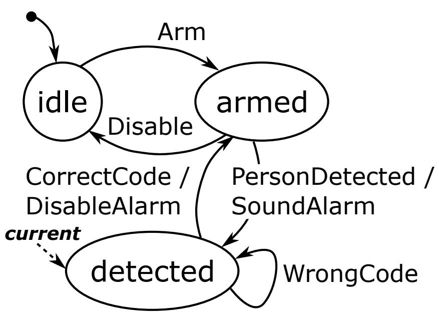
is shown in Figure 6.8a, where a simple home security alarm system is modelled. In the *idle* mode, the alarm system can be armed by the user. If someone is detected in the *armed* mode, the alarm goes off, until the user inputs the correct combination. The alarm can be disabled by sending the *Disable* event, but only when no intrusion is detected. Figure 6.8b presents an example CBD model, modelling the following equations:

$$\begin{cases} y(t) = x(t) + w(t) \\ w(t) = -y(t) \\ z(t) = \begin{cases} x(t) & \text{if } t = 0 \\ y(t-1) & \text{if } t > 0 \end{cases} \end{cases} \tag{6.1}$$

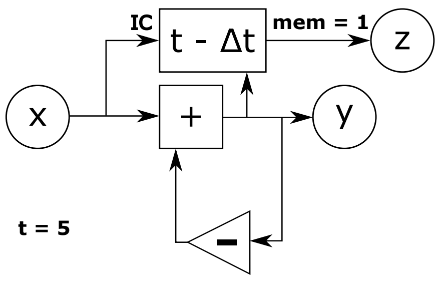
The equation for y is reduced to $y = x - y$, which is a direct feedback loop (termed “algebraic loop”). The CBD formalism handles linear algebraic loops natively, solving $y = x - y$ automatically and generating the necessary code.

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 both the FSA and CBD languages, the partial runtime models are equivalent to the design models, since both languages have operational semantics (they are not compiled).

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 6.9, the full runtime models of the running examples are shown. For FSAs (Figure 6.9a), 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 CBDs (Figure 6.9b), more runtime information is added, as they have a notion of time, represented by the number of iterations. The t variable 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,



(a) The full runtime model of the example FSA model, during execution.



(b) The full runtime model of the example CBD model, during execution.

Figure 6.9: The full runtime models of the examples.

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. Other exceptions are the integration and derivation blocks, for which similar data needs to be remembered and initialized.

Operations

Second, we transpose the various operations on these artefacts: retyping (compilation), simulation (execution), modification (modification), 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 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 language and CBD language do not contain additional information. The retyping operation is therefore trivial in this case.

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 CBD language, 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. Note that, as we are operating on models, and not on generated code, we do not need to consider the technical aspects of executing a newly generated piece of code: the model is updated in-place and the simulation algorithm picks up these changes in the next step.

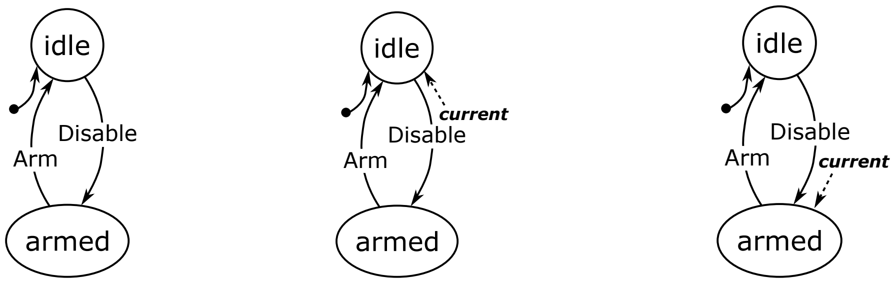
The **Modification** operation is the equivalent of the *modification* operation in programming. Similar to modification in the programming domain, users can only modify the design model. Since all other artefacts are automatically generated, the design model is the only artefact they are familiar with. While the user never edits the partial or full runtime models directly, the design model can be freely modified. 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 6.10a. For the CBD language, users can instantiate new blocks, delete existing blocks, add or remove dependencies, and so on. A modified CBD design model is shown in Figure 6.11a. For both languages, the design models must conform after the modifications.

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 [23]. 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.

The sanitization operation is largely dependent on the kind of state to be merged (*i.e.*, implicit or explicit), 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. In this subsection, we present the sanitization operations for both types of state (*i.e.*, explicit and implicit state), using our running example: the FSA and CBD formalisms. For both, we present realtime live modelling. Note that, similar to live programming, sanitization can only happen when the state is consistent (*i.e.*, after an execution step has ended and before the next execution step has started).

For **explicit state**s, the user can manipulate the execution state by altering the design model. While only the full runtime model has knowledge of which state is the current state (as there is no notion of "execution" in the design model), it is possible to remove the state in the design model that corresponds to the "current" state in the full runtime model. In that case, the state of the running system after the merge is undefined. Changes to any other aspect of the design model do not affect the state of the running system, and are trivially reproduced in the full runtime model.

Resolving an undefined current state is the core task of the sanitization operation. There are three options: reset the explicit state to the initial state, prompt the user for a new value,



(a) Modified design model.

(b) Automatic resolution.

(c) Manual resolution to "armed" state.

Figure 6.10: Sanitization in FSA models.

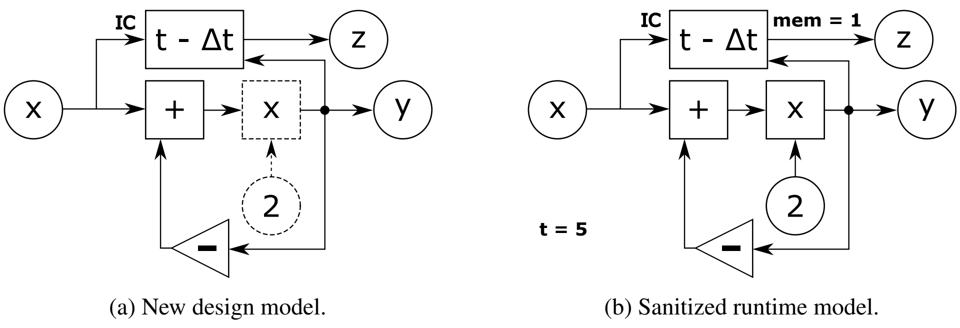
ALGORITHM 11: The FSA sanitize operation.

```
Function SanitizeFSA($M_P^{new}$, $M_F^{old}$)
    if $isInitialized()$ then
        $currState \leftarrow getCurrentState(M_F^{old})$;
        if not $currState \in M_P^{new}$ then
            if $automaticResolution$ then
                $currState \leftarrow getInitialState(M_P^{new})$;
            else
                if $disallowChange$ then
                    raise $Exception$;
                else
                    $currState \leftarrow userChoice(M_P^{new})$;
                end
            end
        end
    else
        $currState \leftarrow initializeState(M_P^{new})$;
    end
end
```

or disallow the change completely. For the new design model in Figure 6.10a and the old full runtime model in Figure 6.9a, the first two options are presented. Figure 6.10b shows automatic resolution where, in this case, the system chooses the default state (the "idle" state) as the new current state. Figure 6.10c shows manual resolution, where the user chooses the "armed" state as the new current state. Algorithm 11 shows the pseudocode of the sanitize operation for FSAs.

Changes resulting in an undefined current state could also be explicitly disallowed. We did not pursue the direction of disallowing design model changes, however, as we explicitly want all modifications to be possible.

For **implicit state**s, the user has no direct access to the execution state, as it is distributed over the model. In contrast to explicit state modelling languages, where the state is either completely removed or completely untouched, we now must keep the state consistent across all modifications. In all cases, however, the sanitization algorithm can manage all changes



(a) New design model. (b) Sanitized runtime model.

Figure 6.11: Sanitization in CBD models.

automatically: enough information is retained to be able to make corrections to the new state.

In our example CBD language, only operations on the integration, derivation, and delay blocks have any influence on the implicit state. Since each block and connection has its own *signal* and *memory*, 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 12: The CBD sanitize operation.

```
Function SanitizeCBD($M_P^{new}, M_F^{old}$)
    forall $block \in M_P^{new}$ 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
    end
    if $isInitialized()$ then
        $iterations \leftarrow getNumberOfIterations(M_F^{old})$;
        $setNumberOfIterations(M_P^{new}, iterations)$;
    else
        $initializeNumberOfIterations(M_P^{new})$;
    end
end
```

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 CBD, 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 CBDs is shown in Algorithm 12.

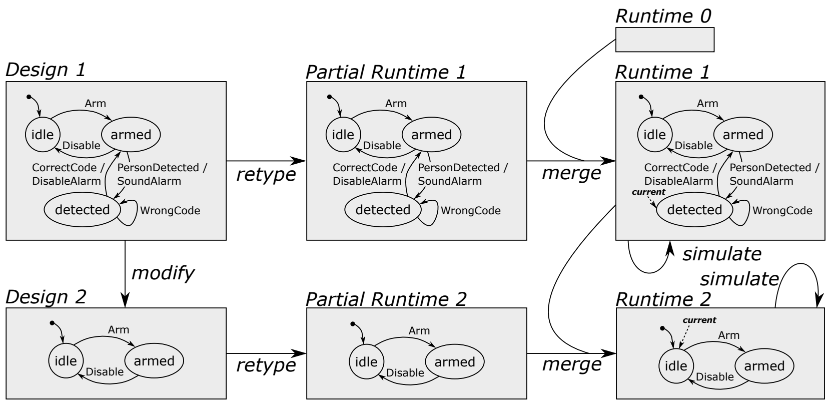


Figure 6.12: Overview of our approach applied to an FSA model.

An example of sanitization is shown in Figure 6.11. In this figure, we see the new design model in Figure 6.11a, and the resulting full runtime model in Figure 6.11b. 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.

Workflow

Figure 6.12 presents an overview of the approach, applied to an FSA model. More generally, Figure 6.13 shows an FTG+PM model describing both the different formalisms and processes of live modelling for any executable modelling language. Noteworthy is the sanitize operation, which has a dual colour: it is mostly automatic, though it can be manual for explicit states when the user is prompted. In the process models, simulation and modification run concurrently: modifications can be made throughout simulation. This is typical for live modelling, in contrast to the mostly linear development process of a single model in ordinary modelling.

6.2.4 Examples

To assess feasibility of our approach, we implemented live modelling for the two running examples. Our prototype consists of two visual modelling and simulation front-ends: one for the FSA language, and one for the CBD language. All operations are defined in the Modelverse [207], our metamodelling tool. The environments allow designing and simulating models in these languages. The result of simulation (*i.e.*, the state of the system) is plotted in the environment in real-time. We reused existing simulators and augmented them with pause and resume operations. Then, we implemented the sanitize operation as described in the previous section. Even during simulation, users can edit the design model in exactly the same way as if simulation was not running: all operations are allowed. When the design model is changed, the changes are sent to the simulator, which first checks

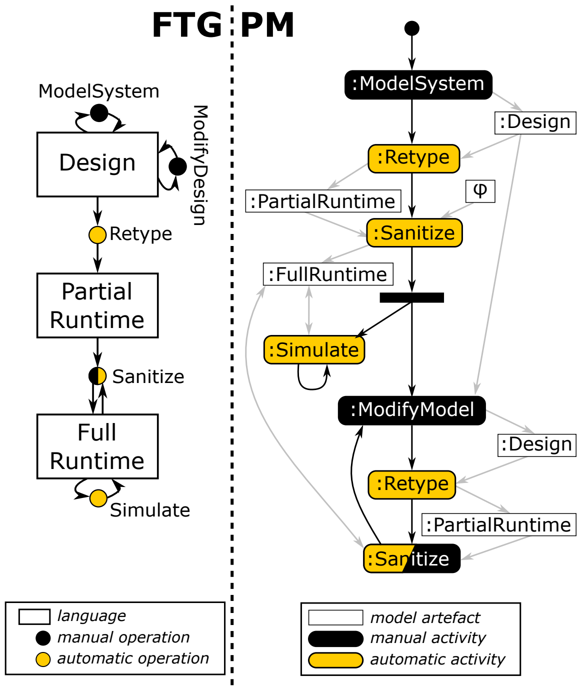


Figure 6.13: Overview of our approach, as an FTG+PM model.

whether the model is valid. If it is not, an error message is shown and the user has to fix the model before continuing simulation. If it is, the sanitize operation is executed, and the user can resume execution.

The implementation of our FSA live modelling environment is shown in Figure 6.14. There are two views: the model editing view (left), and the model simulation view (right). In the editing view, the visual representation of the design model can be modified. The model simulation view shows three traces: the timed input events, the current state throughout time, and the timed output events. Although FSA models are untimed, input events can be raised by the user to the model. 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.

Updates to the design model only influence the future of the simulation run. They can be applied at any point in time, even if simulation is not paused. Each edit that results in a valid model produces an intermediate model. If simulation is running while making edits, simulation continues on these intermediate models as well. When changes break the conformance of the model (*e.g.*, a delay block without initial condition), simulation is automatically paused as it cannot continue in a non-conforming configuration. The user is notified of this problem and will be able to continue simulation as soon as the model conforms again. This guarantees that all subsequent operations happen on a conforming model, and therefore guarantees a consistent model as input for the sanitization operation.

Our implementation of the CBD live modelling environment is shown in Figure 6.15. It is

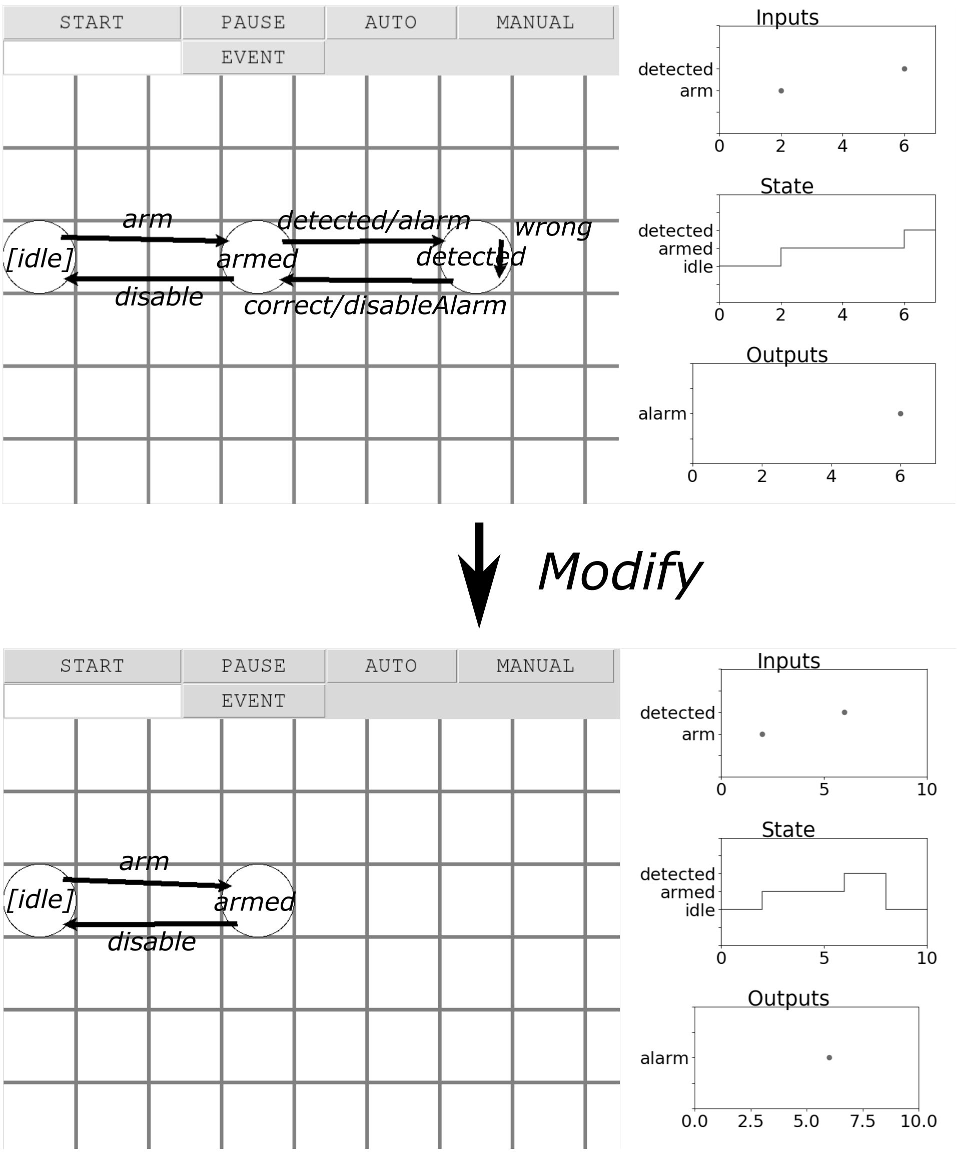


Figure 6.14: Implementation of live modelling for FSAs.

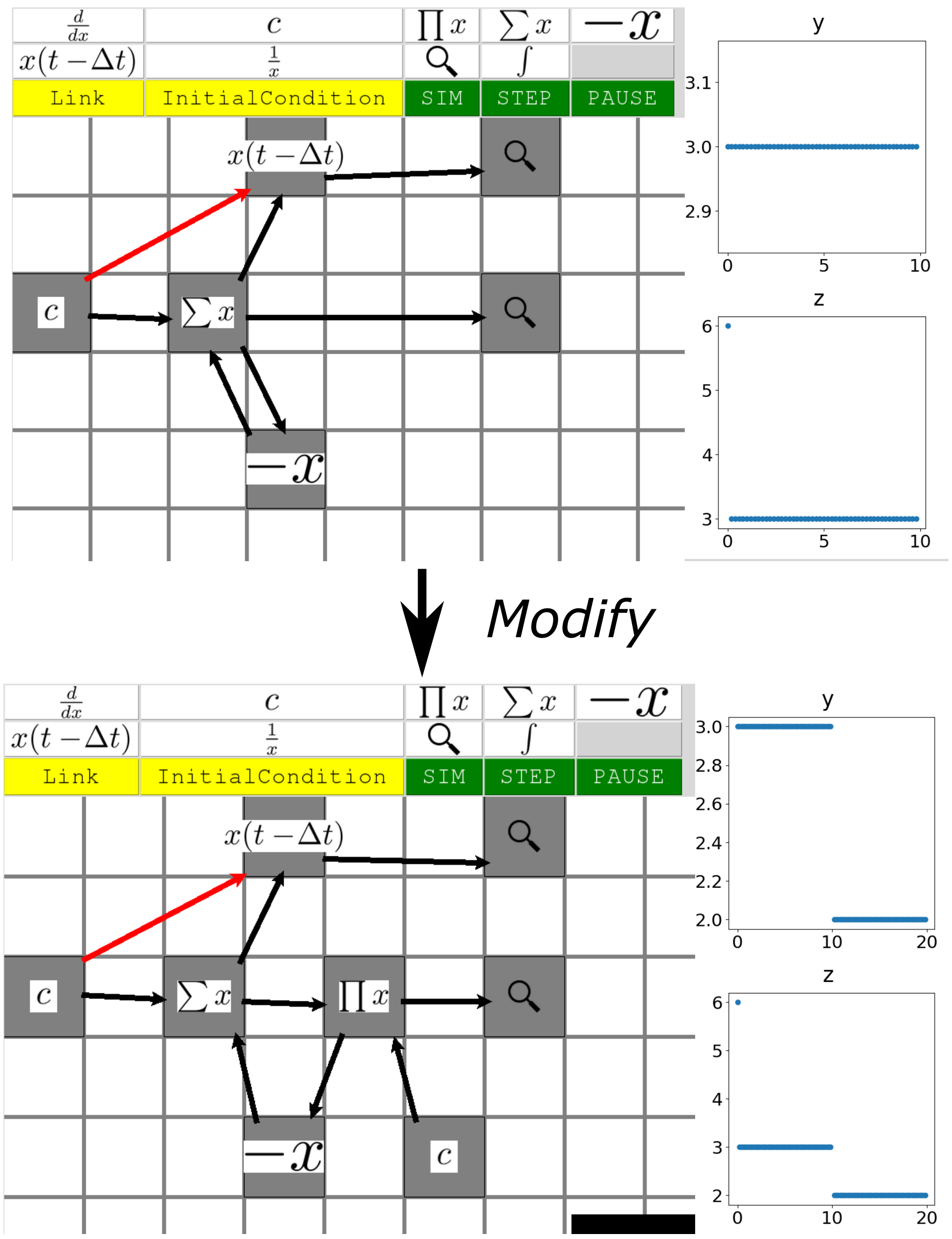


Figure 6.15: Implementation of live modelling for CBDs.

very similar in design to the interface for FSA, and indeed they share a large portion of their code. Again, the model edit view is on the left, and the model simulation view is on the right. When changes are made to the model, they are immediately propagated to the running simulation. Plots show the value of a “probed” signal (corresponding to a block with a magnifying glass icon in the model). It is therefore possible for plots to appear or disappear throughout simulation, when a probe block is added/removed during simulation. This is a design consideration of the UI if it wants to support live modelling.

Summary

This chapter discusses two techniques: omniscient debugging, which allows a user to step back in the simulation trace, and live modelling, which allows users to adapt the design model at runtime, after which the new design is used for the remainder of simulation. Omniscient debugging is usually not memory-efficient, since it needs to keep track of the full simulation trace in order to step back to arbitrary points in that trace. We presented a method that is rooted in optimistic simulation algorithms for keeping the memory and performance overhead low, while still allowing to step back to arbitrary points in the trace. For live modelling, we deconstructed the live programming approach into its artefacts and operations, and transposed those onto modelling formalisms. A workflow allows language engineers to make their languages live based on our generic techniques.

Chapter 7

Conclusion

This thesis set out to contribute to the state-of-the-art in language engineering and modelling tool construction by providing techniques for developing model debugging environments.

We motivate this research by investigating the state-of-the-art in debugging (software) systems. We note that, as software complexity rises, traditional debugging approaches developed for procedural, sequential code are no longer sufficient. When the code is concurrent, or deployed on a parallel or distributed platform, subtle (non-deterministic) synchronization errors can occur. Additionally, software increasingly interacts with an environment, and has to control hardware components. These embedded system are increasingly complex: they can exhibit concurrent, non-deterministic, and continuous behaviour.

Researchers have attempted to overcome the limitations of sequential debuggers by combining multiple debuggers for each thread of control to coordinate them. Quickly, though, such approaches fail to provide efficient debugging support due to the sheer number of threads that need to be controlled and visualized, their complex communication patterns, and behavioural changes in the code due to instrumentation for debugging. Debuggers often have to provide alternate views of system execution by hiding certain details and presenting the results at a more abstract level. Other techniques analyse the program to partly automate the debugging process. Such techniques are, however, not often used by developers, who still prefer to interactively debug their systems.

Model-Driven Engineering (MDE) is a system development method which focuses on the essential complexity of a problem, instead of on how the system is implemented. Models are first-class citizens, which provide a higher level of abstraction. System developers choose a modelling language that is most appropriate to design the different aspects of the system. These models are not guaranteed to be without failures (which means one of their properties was not satisfied), and providing debugging support for them is crucial, taking into account their semantic properties.

The first contribution of this thesis is a technique, called the *de- and reconstruction* of simulators, to instrument existing simulators (implemented in program code) with debugging support. The technique builds an explicit model of the control flow of the simulator, and at that higher level of abstraction, facilitates the task of instrumenting its behaviour with the often complex debugging behaviour. The instrumented model reflects at what points in the simulation algorithm the user can interrupt it, and the hierarchy between different types of "steps" (big steps, small steps, etc.). We identified this "stepping behaviour" as a necessary condition for debugging, since a step clearly delineates at which points the simulation is in a consistent state (and identifies which debugging operations are enabled). A crucial difference in the way semantics is defined was identified as well: in case the semantics are defined operationally, the simulator's definition reflects its dynamics, and the instrumentation can be performed naturally. On the other hand, in case the semantics are defined by mapping onto a formalism with known semantics (translational semantics), this operational view can no longer be deduced from its definition. We developed a method that views the simulator as a "grey box", whose interface exposes the stepping behaviour as if it was implemented operationally. Behind the screen, however, the semantics are implemented translationally. We developed a method to translate domain-specific debug operations to target-level operations, and to translate target-level traces to domain-specific traces. This enables the debugging of formalism whose semantics are defined translationally.

The second contribution is a workflow and architecture, which guide language engineers that want to create complex, often visual, debugging environments for their language. We identified three key components in the architecture of a model debugger:

- The instrumented simulator, which accepts events that influence the execution of the simulation algorithm, and returns events that communicate in which state the simulation is.
- A debugging interface, which is an extension of the runtime interface for the language. It shows the current state of the simulation and allows the user to communicate with the simulator using a toolbar.
- A model-specific visualization, which shows the state of the simulation using a user-specified visualization, and can be instrumented with model-specific debugging support.

These three components are connected by a communication interface, which translates output events from each component to input events for other components. The workflow, lastly, guides language engineers that want to enhance their language with debugging support. It consists of a number of phases, and a number of artefacts that are created in these phases.

The third contribution demonstrates feasibility, by building model debugging environments for a diverse set of languages. We first presented a classification of languages based on their semantic properties. We then assembled a set of representative formalisms, whose semantic features cover the identified semantics properties. We applied our techniques to build debugging environments for six general-purpose modelling languages (whose semantics are defined operationally): action language, CBD, Parallel DEVS, Statecharts, Petrinets, and Dynamic Structure DEVS. We presented a modified technique for building hybrid debuggers as combinations of two debuggers and apply this to a Hybrid Automata formalism. We developed an extension to our technique for domain-specific languages whose semantics are defined translationally, by mapping onto a formalism with known semantics (and for which a debugger exists). By building debugging environments for such a diverse set of modelling languages, we demonstrated that our approach is generally applicable.

The fourth contribution focuses on two advanced debugging techniques: live modelling (a transposition of live programming) and omniscient model debugging. To implement these debugging operations, we similarly provided a workflow, an architecture, and techniques that are generic, such that they are applicable to any modelling language.

Our contributions can be used in multiple areas of future work. We highlight a number of them in this conclusion.

Improving Parallel, Distributed, Code Debuggers

The techniques presented in this thesis take a step back and generalize debugging for languages with varying semantics. They allow system developers to debug the system at the most appropriate level of abstraction, using the abstractions the program was specified with. Most systems are, however, specified in a programming language. As debugging orthogonal, parallel, and distributed systems is challenging with the current level of debugging support in traditional code debuggers, a potential for future work could be to improve debugging techniques for these systems using the techniques presented in this thesis.

ALGORITHM 13: A short program that can be run in parallel.

```
Input: Semaphores $S_1, S_2$
$acquire(S_1)$;
$do\_critical\_section_1()$;
$release(S_1)$; $acquire(S_2)$;
$do\_critical\_section_2()$;
$release(S_2)$;
```

To illustrate how this could work, Algorithm 13 presents an example parallel program, adapted from [99]. A synthetic algorithm is defined with two critical sections: the functions $do\_critical\_section_1()$ and $do\_critical\_section_2()$. We assume this algorithm can be deployed to multiple threads, and the semaphores ensure no two threads will enter a critical section at the same time. To debug such systems efficiently, its inherent non-determinism needs to be taken into account: from one run to the other, a different interleaving of acquires, releases, and executions of critical sections is possible. In their paper [99], Kobler et al. define a way of "swapping" a particular ordering of acquire-release cycles in an execution trace, to check whether the system behaves differently.

A system developer, however, might be interested in *all* execution paths, and the ability to explore these execution paths interactively. Also, within one execution paths, the developer might be interested to *step inside* the actual code executing in the critical section(s). This is an application of two languages for which we implemented debugging techniques in this thesis. On the one hand, Petrinets can be used to model the non-determinism of the concurrent program. On the other hand, action code needs to be modelled that implements the code for the critical section. This means the Petrinets debugger needs to be combined with an action language debugger (in an "embedding" way, as we have seen in Section 5.7) to model the complete system behaviour. Figure 7.1 presents a possible Petrinets model

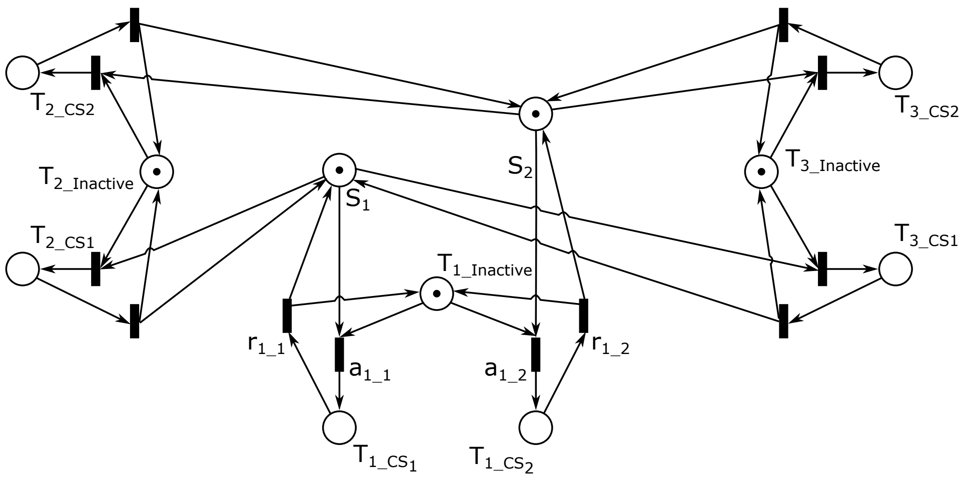


Figure 7.1: A Petrinets model of a program with three threads and two semaphores.

for the concurrent program of Algorithm 13, deployed onto three threads. Two places S_1 and S_2 model the semaphores: if a token is present, the semaphore is not acquired. The three threads are modelled using three places each: $T_{x\_Inactive}$ models whether thread x is active or not, T_{x_CS1} models whether thread x is currently executing the first critical section, and T_{x_CS2} models whether thread x is currently executing the second critical section. Acquiring a semaphore is necessary to start executing a critical section. After a critical section is executed, the semaphore is released by returning the token. This model can be debugged (and the non-deterministic paths explored) using the debugger presented in Section 5.5. When a critical section is entered (a token is present in the appropriate place), however, the code for that critical section is executed. A hybrid language can be created that combines Petrinets with action language, and a debugging interface can be created as in Section 5.7. The user would then be able to step into the execution to investigate potential bugs.

Validation of Debugging Interfaces

We have demonstrated how interactive debugging environment can be constructed. In the future, these techniques can be used to construct debugging environment for other languages, and in other (commercial) environments. Since these languages and environments have a user base, it is important to evaluate the debugging techniques that were implemented. Several questions can be asked:

- Which debugging operations do the users prefer? For example: a "small step" operation might never be used, or a different type of debugging information might be requested by the users. These debugging operations can be rapidly prototyped using our de- and reconstruction technique.
- How can the debugging process best be presented (graphically) to the user? This has to do with the level of detail, but also the layout of presentation. An example

of a creative layout for visualizing DEVS execution can be found in [122]. Our architecture allows the components involved in the debugging process to be replaced, enhanced, and adapted at will. Ultimately, this architecture enables the construction of a library of reusable components from which simulation and domain experts may pick and choose to build debugging environments.

- How does debugging fit into the overall system development process? How can the transition from the debugging environment back to the design environment be as smooth as possible? This couples the advanced debugging techniques such as live modelling to the interactive debugging techniques presented in this thesis.

These studies can quantify the usefulness of different debuggers. These debuggers can be rapidly prototyped using our techniques.

Hybrid Language Engineering

We discussed hybrid language debugging in one particular case: a combination of a discrete-event formalism with a continuous-time formalism, where one is embedded into the other. Many more possibilities exist, for example the combination of a non-deterministic formalism with an action language, as discussed above. The semantics of these formalisms can be interleaved in more exotic ways, leading to defects that are more difficult to debug. And, the interactions between the formalisms can be more complex as well. If semantic adaptation is necessary, that can involve non-trivial computation. A possible future direction of research can look at these interactions (communication) as well as the possible interleaving of the semantics, and come up with general concepts for debugging such hybrid formalisms.

A hybrid language can be seen as the combination of three language "fragments": the two languages that are combined, and an "interaction language". We already assume the two languages that are combined have debugging support, but the interaction language can have debugging support defined for it as well. To sensibly merge these languages, their simulation algorithms, and their debuggers, is a challenging task.

For performance reasons, the layered structure for the debugger we presented in Section 5.7 is not ideal. Instead, a flattened version of the algorithm, as presented in [141] might be more suited. The instrumentation of this simulation algorithm (in its canonical form) is more complicated, however. Future research could look into these two challenges for hybrid language debugging.

Debugging for Co-Simulation

In co-simulation, such as implemented by the FMI standard [1], a set of black-box models are simulated by a master algorithm. Since there is no way of accessing the implementation of the models (mainly for protecting intellectual property), our approach for instrumenting the simulator with debugging support is no longer applicable. These co-simulation standards, however, define a particular interface for the black-box models that allows the master algorithm to control and observe them to some extent. Implementing debugging with that limited amount of information is challenging, and could be impossible (especially for small step semantics). Research into how the master algorithm can implement debugging operations that control the individual black-box models is needed.

Debugging with Optimizations

Debugging optimized code is challenging. As the code is translated from its high-level specification to its low-level deployable form, it is transformed progressively by rearranging, replacing, or completely removing statements. Maintaining traceability to the original program code is difficult or impossible, which means traditional debuggers cannot be used to debug the code. A possible line of future work can extend the debugging of formalisms with translational semantics, since the optimization transformations are similar to the semantic mapping of a formalism to a (lower-level) formalism. Related work in program debugging can be found in techniques for deoptimizing optimized code (on-stack replacement) [83] for debugging such functions, since optimized code hides certain interrupt points the user might be interested in.

Debugging for Evolving Languages

The de- and reconstruction approach presented in this thesis has been applied to languages whose syntax and semantics are well-known. As such, they can be regarded as static, and de- and reconstructed by modelling them as a Statecharts model. If the language definition evolves, however, the definition of the debugger might be inconsistent: certain syntax elements the debugger expects might no longer be present, or the semantics of the formalism were changed in such a way that makes the debugger's execution of the models invalid. The general problem of a language's evolution and the co-evolution of its artefacts (models, editors, compilers, etc.) has been extensively researched [78, 137]. The co-evolution of the debugger is an area of future work which requires the de- and reconstruction to be (partially) automated. In that case, the semantic definition can change, generating a new version of the formalism's semantics, updating its Statecharts model. This is related to the future work on "language fragments", as there, the semantics are generically modelled in a "canonical form" which clearly distinguishes between macro and micro steps. If such standard representation for the execution semantics of a language is developed, the (automatic) instrumentation with debugging support will be facilitated.

Bibliography

[1] *The Functional Mockup Interface*. `https://www.fmi-standard.org/`. Accessed: 2013-09-12. Cited on page 195.

[2] N. A. Allen, C. A. Shaffer, and L. T. Watson, *Building modeling tools that support verification, validation, and testing for the domain expert*, in Proceedings of the 37th Winter Simulation Conference, WSC '05, Winter Simulation Conference, 2005, pp. 419–426. Cited on page 35.

[3] J. Armstrong, *The development of Erlang*, in Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP '97, New York, NY, USA, 1997, ACM, pp. 196–203. Cited on page 34.

[4] C. Artho, *Iterative delta debugging*, International Journal on Software Tools for Technology Transfer, 13 (2011), pp. 223–246. Cited on page 30.

[5] A. Asghar, A. Pop, M. Sjölund, and P. Fritzson, *Efficient Debugging of Large Algorithmic Modelica Applications*, in Proceedings of MATHMOD 2012 - 7th Vienna International Conference on Mathematical Modelling, 2012. Cited on page 35.

[6] M. Bagherzadeh, N. Hili, and J. Dingel, *Model-level, platform-independent debugging in the context of the model-driven development of real-time systems*, in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, New York, NY, USA, 2017, ACM, pp. 419–430. Cited on page 35.

[7] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden, *Extending a traditional debugger to debug massively parallel applications*, Journal of Parallel and Distributed Computing, 64 (2004), pp. 617 – 628. Cited on page 31.

[8] N. Bandener, C. Soltenborn, and G. Engels, *Extending DMM behavior specifications for visual execution and debugging*, in Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers, 2010, pp. 357–376. Cited on pages 4 and 38.

[9] A. Barišić, V. Amaral, M. Goulão, and B. Barroca, *Quality in use of domain-specific languages: A case study*, in Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU '11, ACM, 2011, pp. 65–72. Cited on page 10.

[10] F. J. BARROS, *Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation*, in Proceedings of the 27th Winter Simulation Conference, Dec. 1995, pp. 781–785. Cited on page 122.

[11] F. J. BARROS, *Modeling formalisms for dynamic structure systems*, ACM Trans. Model. Comput. Simul., 7 (1997), pp. 501–515. Cited on page 122.

[12] ——, *Abstract simulators for the DSDE formalism*, in Proceedings of the 30th Winter Simulation Conference, WSC '98, Los Alamitos, CA, USA, 1998, IEEE Computer Society Press, pp. 407–412. Cited on page 126.

[13] P. C. BATES, *Debugging heterogeneous distributed systems using event-based models of behavior*, ACM Trans. Comput. Syst., 13 (1995), pp. 1–31. Cited on page 31.

[14] B. BERTHOMIEU AND F. VERNADAT, *Time petri nets analysis with TINA*, in Third International Conference on the Quantitative Evaluation of Systems - (QEST'06), Sept 2006, pp. 123–124. Cited on page 36.

[15] J. BÉZIVIN, *In Search of a Basic Principle for Model-Driven Engineering*, Novatica Journal Special issue, V (2004), pp. 1–5. Cited on pages 1 and 8.

[16] A. BLOUIN, B. COMBEMALE, B. BAUDRY, AND O. BEAUDOUX, *Kompren: Modeling and Generating Model Slicers*, Software and Systems Modeling, (2012). Cited on page 4.

[17] B. BOOTHE, *Efficient algorithms for bidirectional debugging*, in Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, 2000, pp. 299–310. Cited on page 33.

[18] F. BOULANGER, C. HARDEBOLLE, C. JACQUET, AND D. MARCADET, *Semantic Adaptation for Models of Computation*, in 11th International Conference on Application of Concurrency to System Design (ACSD), 2011, pp. 153–162. Cited on page 136.

[19] E. BOUSSE, J. CORLEY, B. COMBEMALE, J. GRAY, AND B. BAUDRY, *Supporting efficient and advanced omniscient debugging for xDSMLs*, in Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, New York, NY, USA, 2015, ACM, pp. 137–148. Cited on pages 4, 38, and 166.

[20] E. BOUSSE, T. MAYERHOFER, B. COMBEMALE, AND B. BAUDRY, *A Generative Approach to Define Rich Domain-Specific Trace Metamodels*, in 11th European Conference on Modelling Foundations and Applications (ECMFA), L'Aquila, Italy, July 2015. Cited on page 38.

[21] A. BRAGDON, K. ROWAN, J. JACOBSEN, AND R. DELINE, *Debugger canvas: Industrial experience with the code bubbles paradigm*, in ICSE 2010, International Conference on Software Engineering, June 2010. Cited on page 29.

[22] S. BRESLAV, R. GOLDSTEIN, A. TESSIER, AND A. KHAN, *Towards visualization of simulated occupants and their interactions with buildings at multiple time scales*, in Proceedings of the Symposium on Simulation for Architecture & Urban Design, SimAUD '14, San Diego, CA, USA, 2014, Society for Computer Simulation International, pp. 5:1–5:8. Cited on page 24.

[23] G. BRUNET, M. CHECHIK, S. EASTERBROOK, S. NEJATI, N. NIU, AND M. SABETZADEH, *A manifesto for model merging*, in Proceedings of the 2006 International Workshop on Global Integrated Model Management, GaMMa '06, New York, NY, USA, 2006, ACM, pp. 5–12. Cited on page 182.

[24] J. BRÜNING, M. GOGOLLA, L. HAMANN, AND M. KUHLMANN, *Evaluating and Debugging OCL Expressions in UML Models*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 156–162. Cited on page 37.

[25] C. BUCHANAN AND K. KEEFE, *Simulation debugging and visualization in the Möbius modeling framework*, in Quantitative Evaluation of Systems, G. Norman and W. Sanders, eds., vol. 8657 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 226–240. Cited on page 39.

[26] P. BUNUS AND P. FRITZSON, *A debugging scheme for declarative equation based modeling languages*, in Practical Aspects of Declarative Languages, S. Krishnamurthi and C. Ramakrishnan, eds., vol. 2257 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 280–298. Cited on pages 35 and 164.

[27] P. BUNUS AND P. FRITZSON, *Semi-automatic fault localization and behavior verification for physical system simulation models*, in Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Oct 2003, pp. 253–258. Cited on pages 35 and 164.

[28] S. BURCKHARDT, M. FÄHNDRICH, AND J. KATO, *It's alive! continuous feedback in UI programming*, in Proceedings of PLDI '13, 2013, pp. 95–104. Cited on pages 34 and 175.

[29] P. BURGESS, M. LIVESEY, AND C. ALLISON, *Debugging and dynamic modification of embedded systems*, in Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences, vol. 1, Jan 1996, pp. 489–498 vol.1. Cited on page 33.

[30] M. M. BURNETT, J. W. ATWOOD, JR., AND Z. T. WELCH, *Implementing level 4 liveness in declarative visual programming languages*, in Proceedings of Visual Languages '98, 1998, pp. 126–133. Cited on page 34.

[31] C. CAERTS, R. LAUWEREINS, AND J. PEPERSTRAETE, *PDG: a process-level debugger for concurrent programs in the GRAPE rapid prototyping environment*, in Rapid System Prototyping, 1993. Shortening the Path from Specification to Prototype. Proceedings., Fourth International Workshop on, Jun 1993, pp. 17–30. Cited on page 31.

[32] D. ÇETINKAYA, A. VERBRAECK, AND M. D. SECK, *Model continuity in discrete event simulation: A framework for model-driven development of simulation models*, ACM Trans. Model. Comput. Simul., 25 (2015), pp. 17:1–17:24. Cited on page 23.

[33] F. E. CELLIER, *Continuous system modeling*, Springer-Verlag, New York, 1991. Cited on pages 2, 8, 13, and 79.

[34] S. CHANDRA, E. TORLAK, S. BARMAN, AND R. BODIK, *Angelic debugging*, in Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, New York, NY, USA, 2011, ACM, pp. 121–130. Cited on page 30.

[35] R. N. Charette, *This Car Runs on Code*, IEEE Spectrum, (2009). Cited on pages 1 and 7.

[36] Y.-P. Cheng, J.-F. Chen, M.-C. Chiu, N.-W. Lai, and C.-C. Tseng, *xDIVA: A debugging visualization system with composable visualization metaphors*, in Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA Companion '08, New York, NY, USA, 2008, ACM, pp. 807–810. Cited on page 29.

[37] A. Chiş, M. Denker, T. Gîrba, and O. Nierstrasz, *Practical domain-specific debuggers using the moldable debugger framework*, Comput. Lang. Syst. Struct., 44 (2015), pp. 89–113. Cited on page 38.

[38] A. C. Chow, *Parallel DEVS: A parallel, hierarchical, modular modelling formalism and its distributed simulator*, Transactions of the SCS, 13 (1996), pp. 55–67. Cited on pages 88 and 94.

[39] J. Cleary, F. Gomes, B. Unger, Z. Xiao, and R. Thudt, *Cost of state saving & rollback*, SIGSIM Simululation Digest, 24 (1994), pp. 94–101. Cited on page 168.

[40] J. Corley, *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 36.

[41] ——, *Exploring omniscient debugging for model transformations*, in Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 28 - October 3, 2014., 2014, pp. 63–68. Cited on page 36.

[42] J. Corley, B. P. Eddy, and J. Gray, *Towards efficient and scalabale omniscient debugging for model transformations*, in Proceedings of the 14th Workshop on Domain-Specific Modeling, DSM '14, ACM, 2014, pp. 13–18. Cited on page 36.

[43] J. Corley, B. P. Eddy, E. Syriani, and J. Gray, *Efficient and scalable omniscient debugging for model transformations*, Software Quality Journal, (2016), pp. 1–42. Cited on pages 4, 36, and 166.

[44] J. H. Cross, II, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, and L. N. Montgomery, *Robust generation of dynamic data structure visualizations with multiple interaction approaches*, Trans. Comput. Educ., 9 (2009), pp. 13:1–13:32. Cited on page 29.

[45] J. Cunha, J. Lourenço, J. Vieira, B. *Moscão*, and D. Pereira, *A framework to support parallel and distributed debugging*, in High-Performance Computing and Networking, P. Sloot, M. Bubak, and B. Hertzberger, eds., vol. 1401 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1998, pp. 708–717. Cited on page 31.

[46] E. Czaplicki, *Elm: Concurrent FRP for functional GUIs*. `https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf`, 2012. Cited on pages 34, 176, and 177.

[47] V. Djukić, A. Popović, and Z. Lu, *Run-time code generators for model-level debugging in domain-specific modeling*, in Proc. DSM, ACM, 2016, pp. 1–7. Cited on page 164.

[48] D. Dotan and A. Kirshin, *Debugging and testing behavioral UML models*, in Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, OOPSLA '07, New York, NY, USA, 2007, ACM, pp. 838–839. Cited on page 37.

[49] J. Edwards, *Subtext: Uncovering the simplicity of programming*, in Proceedings of OOPSLA '05, 2005, pp. 505–518. Cited on page 34.

[50] J. Engblom, *A review of reverse debugging*, in System, Software, SoC and Silicon Debug Conference (S4D), 2012, Sept 2012, pp. 1–6. Cited on page 33.

[51] S. Esmaeilsabzali, N. A. Day, J. M. Atlee, and J. Niu, *Deconstructing the semantics of big-step modelling languages*, Requirements Engineering, 15 (2010), pp. 235–265. Cited on pages 20 and 105.

[52] R. Ewald and A. M. Uhrmacher, *SESSL: A domain-specific language for simulation experiments*, ACM Transactions on Modeling and Computer Simulation, 24 (2014), pp. 11:1–11:25. Cited on page 23.

[53] R. S. Fabry, *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 page 34.

[54] E. C. Freuder, R. J. Wallace, and T. E. Nordlander, *Debugging constraint models with metamodels and metaknowledge*, Constraint Modelling and Reformulation (ModRef09), (2009), p. 45. Cited on page 38.

[55] P. Fritzson and P. Bunus, *Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation*, in Proceedings 35th Annual Simulation Symposium. SS 2002, April 2002, pp. 365–380. Cited on pages 35 and 46.

[56] E. Fromentin, N. Plouzeau, and M. Raynal, *An introduction to the analysis and debug of distributed computations*, in Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP., IEEE First International Conference on, vol. 2, Apr 1995, pp. 545–553 vol.2. Cited on page 31.

[57] L. Fuentes, J. Manrique, and P. Sánchez, *Pópulo: A tool for debugging UML models*, in Companion of the 30th International Conference on Software Engineering, ICSE Companion '08, New York, NY, USA, 2008, ACM, pp. 955–956. Cited on page 37.

[58] R. M. Fujimoto, *Parallel discrete event simulation*, Communications of the ACM, (1990), pp. 30–53. Cited on page 167.

[59] ——, *Parallel and Distribution Simulation Systems*, John Wiley & Sons, Inc., 1st ed., 1999. Cited on pages 167 and 168.

[60] S. Gabmeyer, P. Kaufmann, M. Seidl, M. Gogolla, and G. Kappel, *A feature-based classification of formal verification techniques for software models*, Software & Systems Modeling, (2017). Cited on page 2.

[61] GDB, *GDB reversible debugging*. `https://www.gnu.org/software/gdb/news/reversible.html`, 2009. Cited on page 33.

[62] GDB DEVELOPERS, *Gdb: The GNU project debugger*. `https://www.gnu.org/software/gdb/news/reversible.html`, 2017. Cited on page 28.

[63] L. GEIGER AND A. ZNDORF, *Graph based debugging with fujaba*, Electronic Notes in Theoretical Computer Science, 72 (2002), p. 112. Cited on page 36.

[64] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. Cited on pages 34 and 177.

[65] C. GOMES, *Foundations for Continuous Time Hierarchical Co-simulation*, in ACM Student Research Competition (ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems), Saint Malo, Brittany, France, 2016, p. to appear. Cited on page 136.

[66] R. GORE, P. F. REYNOLDS JR., D. KAMENSKY, S. DIALLO, AND J. PADILLA, *Statistical debugging for simulations*, ACM Transactions on Modeling and Computer Simulation, 25 (2015), pp. 16:1–16:26. Cited on page 39.

[67] P. GRAF AND K. MÜLLER-GLASER, *Dynamic mapping of runtime information models for debugging embedded software*, in Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on, June 2006, pp. 3–9. Cited on page 37.

[68] P. GRAF, C. REICHMANN, AND K. D. MÜLLER-GLASER, *Towards a Platform for Debugging Executed UML-Models in Embedded Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 238–241. Cited on page 37.

[69] H. GRÖNNIGER, H. KRAHN, B. RUMPE, M. SCHINDLER, AND S. VÖLKEL, *Text-based modeling*, in Proceedings of the 4th International Workshop on Software Language Engineering, 2007. Cited on page 34.

[70] E. GUNTER AND D. PELED, *Temporal debugging for concurrent systems*, in Tools and Algorithms for the Construction and Analysis of Systems, J.-P. Katoen and P. Stevens, eds., vol. 2280 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 431–444. Cited on page 32.

[71] B. HAILPERN AND P. SANTHANAM, *Software debugging, testing, and verification*, IBM Systems Journal, 41 (2002), pp. 4–12. Cited on page 27.

[72] C. M. HANCOCK, *Real-Time Programming and the Big Ideas of Computational Literacy*, PhD thesis, Massachusetts Institute of Technology, 2003. Cited on page 34.

[73] D. HAREL, *Statecharts: a visual formalism for complex systems*, Science of Computer Programming, 8 (1987), pp. 231–274. Cited on pages 2, 8, 13, and 17.

[74] D. HAREL AND H. KUGLER, *The Rhapsody semantics of Statecharts (or, on the executable core of the UML)*, in Integration of Software Specification Techniques for Applications in Engineering, vol. 3147 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 325–354. Cited on page 20.

[75] D. HAREL AND A. NAAMAD, *The STATEMATE Semantics of Statecharts*, ACM Trans. Softw. Eng. Methodol., 5 (1996), pp. 293–333. Cited on page 20.

[76] D. HAREL, A. PNUELI, J. P. SCHMIDT, AND R. SHERMAN, *On the formal semantics of Statecharts*, in Proceedings of the 2nd IEEE Symposium on Logic in Computer Science, 1987, pp. 54–64. Cited on page 19.

[77] D. HAREL AND B. RUMPE, *Meaningful modeling: What's the semantics of "semantics"?*, Computer, 37 (2004), pp. 64–72. Cited on page 9.

[78] R. HEBIG, D. E. KHELLADI, AND R. BENDRAOU, *Approaches to co-evolution of metamodels and models: A survey*, IEEE Transactions on Software Engineering, 43 (2017), pp. 396–414. Cited on page 196.

[79] A. HEGEDUS, G. BERGMANN, I. RATH, AND D. VARRO, *Back-annotation of simulation traces with change-driven model transformations*, in 8th IEEE International Conference on Software Engineering and Formal Methods, 2010, pp. 145–155. Cited on pages 40 and 149.

[80] M. HIBBERD, M. LAWLEY, AND K. RAYMOND, *Forensic debugging of model transformations*, in Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, MODELS'07, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 589–604. Cited on page 36.

[81] F. HILKEN, M. GOGOLLA, L. BURGUEÑO, AND A. VALLECILLO, *Testing models and model transformations using classifying terms*, Software & Systems Modeling, (2016). Cited on page 2.

[82] K. HINDRIKS, *Debugging is explaining*, in PRIMA 2012: Principles and Practice of Multi-Agent Systems, I. Rahwan, W. Wobcke, S. Sen, and T. Sugawara, eds., vol. 7455 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 31–45. Cited on page 32.

[83] U. HÖLZLE, C. CHAMBERS, AND D. UNGAR, *Debugging optimized code with dynamic deoptimization*, in Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, New York, NY, USA, 1992, ACM, pp. 32–43. Cited on page 196.

[84] G. J. HOLZMANN, *The model checker SPIN*, IEEE Trans. Softw. Eng., 23 (1997), pp. 279–295. Cited on pages 117 and 156.

[85] A. HOPKINS AND K. MCDONALD-MAIER, *Debug support for complex systems on-chip: a review*, Computers and Digital Techniques, IEE Proceedings -, 153 (2006), pp. 197–207. Cited on page 33.

[86] A. JANTSCH AND I. SANDER, *Models of computation and languages for embedded system design*, IEE Proceedings - Computers and Digital Techniques, 152 (2005), pp. 114–129(15). Cited on page 136.

[87] D. R. JEFFERSON, *Virtual time*, ACM Transactions on Programming Languages and Systems, 7 (1985), pp. 404–425. Cited on page 167.

[88] M. JUKŠS, C. VERBRUGGE, AND H. VANGHELUWE, *Transformations debugging transformations*, in Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located

with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), 2017. Cited on page 37.

[89] P. Kacsuk, R. Lovas, and J. Kovács, *Systematic debugging of parallel programs in DIWIDE based on collective breakpoints and macrosteps*, in Euro-Par99 Parallel Processing, P. Amestoy, P. Berger, M. Daydé, D. Ruiz, I. Duff, V. Frayssé, and L. Giraud, eds., vol. 1685 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1999, pp. 90–97. Cited on page 31.

[90] P. Kaufmann, M. Kronegger, A. Pfandler, M. Seidl, and M. Widl, *A SAT-based debugging tool for state machines and sequence diagrams*, in Software Language Engineering, B. Combemale, D. Pearce, O. Barais, and J. Vinju, eds., vol. 8706 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 21–40. Cited on page 37.

[91] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*, John Wiley & Sons, 2008. Cited on pages 2, 8, and 37.

[92] P. Kemper, *A trace-based visual inspection technique to detect errors in simulation models*, in Simulation Conference, 2007 Winter, Dec 2007, pp. 747–755. Cited on page 39.

[93] P. Kemper and C. Tepper, *A Petri net approach to verify and debug simulation models*, in Simulation and Verification of Dynamic Systems, D. M. Nicol, C. Priami, H. R. Nielson, and A. M. Uhrmacher, eds., Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. Cited on page 39.

[94] S. Kent, *Model Driven Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 286–298. Cited on pages 1 and 8.

[95] Y. P. Khoo, J. S. Foster, and M. Hicks, *Expositor: Scriptable time-travel debugging with first-class traces*, in Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, Piscataway, NJ, USA, 2013, IEEE Press, pp. 352–361. Cited on page 29.

[96] G. Kistner and C. Nuernberger, *Developing user interfaces using SCXML Statecharts*, in Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML, D. Schnelle-Walka, S. Radomski, T. Lager, J. Barnett, D. Dahl, and M. Mühlhäuser, eds., 2014, pp. 5–11. Cited on page 23.

[97] A. Kleppe, *A language description is more than a metamodel*, in Fourth International Workshop on Software Language Engineering, 2007. Cited on pages 8 and 10.

[98] A. J. Ko and B. A. Myers, *Debugging reinvented: Asking and answering why and why not questions about program behavior*, in Proceedings of the 30th International Conference on Software Engineering, ICSE '08, New York, NY, USA, 2008, ACM, pp. 301–310. Cited on page 30.

[99] R. Kobler, D. Kranzlmüller, and J. Volkert, *Debugging OpenMP programs using event manipulation*, in OpenMP Shared Memory Parallel Programming,

R. Eigenmann and M. Voss, eds., vol. 2104 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 81–89. Cited on pages 32 and 193.

[100] A. KOCH AND A. ZNDORF, *Graphical debugging of distributed applications: Using UML object diagrams to visualize the state of distributed applications at runtime*, in 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Feb 2015, pp. 223–230. Cited on page 31.

[101] E. KRAEMER AND J. STASKO, *The visualization of parallel systems: An overview*, Journal of Parallel and Distributed Computing, 18 (1993), pp. 105 – 117. Cited on page 31.

[102] D. KRAHL, *Debugging simulation models*, in Simulation Conference, 2005 Proceedings of the Winter, Dec 2005, pp. 7 pp.–. Cited on page 39.

[103] A. KRASNOGOLOWY, S. HILDEBRANDT, AND S. WTZOLDT, *Flexible debugging of behavior models*, in Proceedings of the 2012 IEEE International Conference on Industrial Technology (ICIT), March 2012, pp. 331–336. Cited on page 37.

[104] A. KUHN, G. C. MURPHY, AND C. A. THOMPSON, *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, MODELS'12, Berlin, Heidelberg, 2012, Springer-Verlag, pp. 352–367. Cited on page 175.

[105] T. KÜHNE, *Matters of (meta-) modeling*, Software & Systems Modeling, 5 (2006), pp. 369–385. Cited on pages 10 and 37.

[106] T. KÜHNE, G. MEZEI, E. SYRIANI, H. VANGHELUWE, AND M. WIMMER, *Explicit transformation modeling*, in Proceedings of the 2009 International Conference on Models in Software Engineering, MODELS'09, Berlin, Heidelberg, 2010, Springer-Verlag, pp. 240–255. Cited on page 12.

[107] Y. LAURENT, R. BENDRAOU, AND M.-P. GERVAIS, *Executing and debugging UML models: An fUML extension*, in Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, New York, NY, USA, 2013, ACM, pp. 1095–1102. Cited on pages 4 and 37.

[108] E. A. LEE, *The problem with threads*, Computer, 39 (2006), pp. 33–42. Cited on page 17.

[109] E. A. LEE, *Cyber physical systems: Design challenges*, in 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), May 2008, pp. 363–369. Cited on pages 1 and 8.

[110] R. LENCEVICIUS, *On-the-fly query-based debugging with examples*, in Proceedings Fourth International Workshop on Automated Debugging, 2000. Cited on page 29.

[111] R. LENCEVICIUS, U. HÖLZLE, AND A. K. SINGH, *Query-based debugging of object-oriented programs*, in Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '97, New York, NY, USA, 1997, ACM, pp. 304–317. Cited on page 29.

[112] ——, *Dynamic query-based debugging of object-oriented programs*, Automated Software Engg., 10 (2003), pp. 39–74. Cited on page 29.

[113] M. LESKE, A. CHIŞ, AND O. NIERSTRASZ, *Improving live debugging of concurrent threads through thread histories*, Science of Computer Programming, (2017). Cited on page 31.

[114] H. LIEBERMAN AND C. FRY, *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 176.

[115] M. H. V. LIEDEKERKE AND N. M. AVOURIS, *Debugging multi-agent systems*, Information and Software Technology, 37 (1995), pp. 103 – 112. Cited on page 32.

[116] A. LIENHARD, T. GÎRBA, AND O. NIERSTRASZ, *Practical object-oriented back-in-time debugging*, in Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP '08, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 592–615. Cited on pages 33 and 166.

[117] R. T. LINDEMAN, L. C. KATS, AND E. VISSER, *Declaratively defining domain-specific language debuggers*, in Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering, GPCE '11, New York, NY, USA, 2011, ACM, pp. 127–136. Cited on pages 37 and 164.

[118] A. LUANGSODSAI AND C. FOX, *Concurrent statechart slicing*, in Computer Science and Electronic Engineering Conference (CEEC), 2010 2nd, Sept 2010, pp. 1–7. Cited on page 35.

[119] L. LÚCIO, S. MUSTAFIZ, J. DENIL, B. MEYERS, AND H. VANGHELUWE, *The formalism transformation graph as a guide to Model Driven Engineering*, Tech. Report SOCS-TR-2012.1, McGill University, March 2012. Cited on page 16.

[120] L. LÚCIO, S. MUSTAFIZ, J. DENIL, H. VANGHELUWE, AND M. JUKSS, *FTG+PM: An Integrated Framework for Investigating Model Transformation Chains*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 182–202. Cited on page 16.

[121] C.-K. LUK, R. COHN, R. MUTH, H. PATIL, A. KLAUSER, G. LOWNEY, S. WALLACE, V. J. REDDI, AND K. HAZELWOOD, *Pin: Building customized program analysis tools with dynamic instrumentation*, in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, New York, NY, USA, 2005, ACM, pp. 190–200. Cited on page 29.

[122] M. MALEKI, R. WOODBURY, R. GOLDSTEIN, S. BRESLAV, AND A. KHAN, *Designing DEVS visual interfaces for end-user programmers*, SIMULATION, 91 (2015), pp. 715–734. Cited on pages 36 and 195.

[123] R. MANNADIAR AND H. VANGHELUWE, *Debugging in domain-specific modelling*, in Software Language Engineering, B. Malloy, S. Staab, and M. van den Brand, eds., vol. 6563 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 276–285. Cited on page 37.

[124] G. MARCEAU, G. H. COOPER, J. P. SPIRO, S. KRISHNAMURTHI, AND S. P. REISS, *The design and implementation of a dataflow language for scriptable debugging*, Automated Software Engg., 14 (2007), pp. 59–86. Cited on page 29.

[125] M. Martin, B. Livshits, and M. S. Lam, *Finding application errors and security flaws using PQL: A Program Query Language*, in Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, New York, NY, USA, 2005, ACM, pp. 365–383. Cited on page 29.

[126] W. Mayer and M. Stumptner, *Model-based debugging – state of the art and future challenges*, Electron. Notes Theor. Comput. Sci., 174 (2007), pp. 61–82. Cited on page 30.

[127] W. Mayer and M. Stumptner, *Evaluating models for model-based debugging*, in Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, Washington, DC, USA, 2008, IEEE Computer Society, pp. 128–137. Cited on page 30.

[128] T. Mayerhofer, *Testing and debugging UML models based on fUML*, in Proceedings of the 34th International Conference on Software Engineering, ICSE '12, Piscataway, NJ, USA, 2012, IEEE Press, pp. 1579–1582. Cited on pages 4 and 37.

[129] S. McDirmid, *Living it up with a live programming language*, in Proceedings of OOPSLA '07, 2007, pp. 623–638. Cited on page 34.

[130] ——, *Usable live programming*, in Proceedings of Onward! 2013, 2013, pp. 53–61. Cited on pages 34 and 176.

[131] C. E. McDowell and D. P. Helmbold, *Debugging concurrent programs*, ACM Comput. Surv., 21 (1989), pp. 593–622. Cited on page 30.

[132] K. Mehner, *JaVis: A UML-based visualization and debugging environment for concurrent Java programs*, in Software Visualization, S. Diehl, ed., vol. 2269 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 163–175. Cited on page 31.

[133] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Cited on page 37.

[134] T. Mens and P. Van Gorp, *A taxonomy of model transformation*, Electron. Notes Theor. Comput. Sci., 152 (2006), pp. 125–142. Cited on page 11.

[135] T. Mészáros, P. Fehér, and L. Lengyel, *Visual debugging support for graph rewriting-based model transformations*, in Proceedings of Eurocon 2013, International Conference on Computer as a Tool, Zagreb, Croatia, July 1-4, 2013, IEEE, 2013, pp. 482–488. Cited on page 36.

[136] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer, *ProMoBox: A framework for generating domain-specific property languages*, in Software Language Engineering, vol. 8706 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 1–20. Cited on pages 2, 11, 40, 41, and 156.

[137] B. Meyers and H. Vangheluwe, *A framework for evolution of modelling languages*, Science of Computer Programming, 76 (2011), pp. 1223 – 1246. Special Issue on Software Evolution, Adaptability and Variability. Cited on page 196.

[138] P. J. MOSTERMAN AND H. VANGHELUWE, *Computer automated multi-paradigm modeling: An introduction*, Simulation, 80 (2004), pp. 433–450. Cited on pages 2 and 8.

[139] T. MURATA, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE, 77 (1989), pp. 541–580. Cited on pages 2, 13, and 36.

[140] S. MUSTAFIZ, J. DENIL, L. LÚCIO, AND H. VANGHELUWE, *The FTG+PM framework for multi-paradigm modelling: An automotive case study*, in Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM '12, New York, NY, USA, 2012, ACM, pp. 13–18. Cited on page 16.

[141] S. MUSTAFIZ, C. GOMES, B. BARROCA, AND H. VANGHELUWE, *Modular Design of Hybrid Languages by Explicit Modeling of Semantic Adaptation*, in Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, DEVS '16, San Diego, CA, USA, 2016, pp. 29:1—-29:8. Cited on pages 137, 139, 142, and 195.

[142] S. MUSTAFIZ AND H. VANGHELUWE, *Explicit modelling of Statechart simulation environments*, in Summer Simulation Multiconference, Society for Computer Simulation International (SCS), July 2013, pp. 445 – 452. Cited on pages 35 and 53.

[143] A. MUZY AND J. J. NUTARO, *Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators*, in 1st Open International Conference on Modeling & Simulation (OICMS), 2005, pp. 273–279. Cited on pages 94 and 126.

[144] S. NARAYANASAMY, G. POKAM, AND B. CALDER, *Bugnet: Continuously recording program execution for deterministic replay debugging*, SIGARCH Comput. Archit. News, 33 (2005), pp. 284–295. Cited on page 32.

[145] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, *Software errors cost U.S. economy $59.5 billion annually*, October 2002. Cited on pages 3 and 27.

[146] N. NETHERCOTE AND J. SEWARD, *Valgrind: A framework for heavyweight dynamic binary instrumentation*, in Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, New York, NY, USA, 2007, ACM, pp. 89–100. Cited on page 29.

[147] C. B. NIELSEN, P. G. LARSEN, J. FITZGERALD, J. WOODCOCK, AND J. PELESKA, *Systems of systems engineering: Basic concepts, model-based techniques, and research directions*, ACM Comput. Surv., 48 (2015), pp. 18:1–18:41. Cited on pages 1 and 8.

[148] J. NUTARO, *adevs*. `http://sourceforge.net/projects/adevs/`. Cited on pages 94 and 126.

[149] R. A. OLSSON, R. H. CRAWFORD, AND W. W. HO, *A dataflow approach to event-based debugging*, Softw. Pract. Exper., 21 (1991), pp. 209–229. Cited on page 29.

[150] C. M. PANCAKE, *Graphical support for parallel debugging*, in Software for Parallel Computation, J. Kowalik and L. Grandinetti, eds., vol. 106 of NATO ASI Series, Springer Berlin Heidelberg, 1993, pp. 216–228. Cited on page 31.

[151] C. M. Pancake and S. Utter, *Models for visualization in parallel debuggers*, in Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing '89, New York, NY, USA, 1989, ACM, pp. 627–636. Cited on page 31.

[152] T. Park and P. I. Barton, *State event location in differential-algebraic models*, ACM Trans. Model. Comput. Simul., 6 (1996), pp. 137–165. Cited on page 139.

[153] D. Pavletic and K. Halbauer, *Interactive debugging for extensible languages in multi-stage transformation environments*, in Proceedings of the 2nd International Workshop on Executable Modeling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), T. Mayerhofer, P. Langer, E. Seidewitz, and J. Gray, eds., vol. 1760, CEUR, 2016, pp. 19–25. Cited on page 38.

[154] D. Pavletic, M. Voelter, S. A. Raza, B. Kolb, and T. Kehrer, *Extensible debugger framework for extensible languages*, in Reliable Software Technologies – Ada-Europe 2015: 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings, J. A. de la Puente and T. Vardanega, eds., Cham, 2015, Springer International Publishing, pp. 33–49. Cited on page 38.

[155] M. Petre, *Why looking isn't always seeing: Readership skills and graphical programming*, Commun. ACM, 38 (1995), pp. 33–44. Cited on page 34.

[156] E. Planas, J. Cabot, and C. Gmez, *Lightweight and static verification of UML executable models*, Computer Languages, Systems & Structures, 46 (2016), pp. 66 – 90. Cited on page 2.

[157] A. Pop and P. Fritzson, *A portable debugger for algorithmic Modelica code*, in Proceedings of the 4th International Modelica Conference, G. Schmitz, ed., Mar. 2005. Cited on pages 35 and 164.

[158] A. Pop, M. Sjölund, A. Asghar, P. Fritzson, and C. Francesco, *Static and Dynamic Debugging of Modelica Models*, in Proceedings of the 9th International Modelica Conference, Nov. 2012, pp. 443–454. Cited on page 35.

[159] A. Pop, M. Sjölund, A. Ashgar, P. Fritzson, and F. Casella, *Integrated Debugging of Modelica Models*, Modeling, Identification and Control, 35 (2014), pp. 93–107. Cited on pages 35 and 164.

[160] A. Potanin, J. Noble, and R. Biddle, *Snapshot query-based debugging*, in Proceedings of the 2004 Australian Software Engineering Conference, ASWEC '04, Washington, DC, USA, 2004, IEEE Computer Society, p. 251. Cited on page 29.

[161] G. Pothier and E. Tanter, *Back to the future: Omniscient debugging*, IEEE Software, 26 (2009), pp. 78–85. Cited on page 166.

[162] G. Pothier, E. Tanter, and J. Piquer, *Scalable omniscient debugging*, in Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07, ACM, 2007, pp. 535–552. Cited on pages 33 and 166.

[163] K. Pouget, P. Lopez Cueva, M. Santana, and J.-F. Mehaut, *Interactive debugging of dynamic dataflow embedded applications*, in Parallel and Distributed

Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International, May 2013, pp. 345–354. Cited on page 33.

[164] D. Poutakidis, M. Winikoff, L. Padgham, and Z. Zhang, *Debugging and testing of multi-agent systems using design artefacts*, in Multi-Agent Programming:, A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, eds., Springer US, 2009, pp. 215–258. Cited on page 32.

[165] B. R. Preiss, W. M. Loucks, and I. D. Macintyre, *Effects of the checkpoint interval on time and space in time warp*, ACM Trans. Model. Comput. Simul., 4 (1994), pp. 223–253. Cited on page 168.

[166] W. Pugh, *The Java memory model is fatally flawed*, Concurrency: Practice and Experience, 12 (2000), pp. 445–455. Cited on page 13.

[167] J. Ressia, A. Bergel, and O. Nierstrasz, *Object-centric debugging*, in Software Engineering (ICSE), 2012 34th International Conference on, June 2012, pp. 485–495. Cited on page 29.

[168] F. Rogin and R. Drechsler, *High-level debugging and exploration*, in Debugging at the Electronic System Level, Springer Netherlands, 2010, pp. 71–104. Cited on page 39.

[169] R. Rönngren and R. Ayani, *Adaptive checkpointing in time warp*, SIGSIM Simul. Dig., 24 (1994), pp. 110–117. Cited on page 168.

[170] M. Ronsse, K. De Bosschere, and J. Chassin de Kergommeaux, *Execution replay and debugging*, in Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG 2000), M. Ducassé, ed., Aug. 2000. Cited on page 32.

[171] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*, Pearson Higher Education, 2004. Cited on page 2.

[172] D. A. Sadilek and G. Wachsmuth, *Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 63–78. Cited on page 38.

[173] Y. Saito, *Jockey: A user-space library for record-replay debugging*, in Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, AADEBUG'05, New York, NY, USA, 2005, ACM, pp. 69–76. Cited on page 32.

[174] E. Sandewall, *Programming in an interactive environment: The "Lisp" experience*, ACM Comput. Surv., 10 (1978), pp. 35–71. Cited on page 34.

[175] J. Schönbčk, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger, and W. Schwinger, *Tetrabox - a generic white-box testing framework for model transformations*, in 2013 20th Asia-Pacific Software Engineering Conference (APSEC), vol. 1, Dec 2013, pp. 75–82. Cited on page 4.

[176] J. Schönböck, *Testing and Debugging of Model Transformations*, PhD thesis, E188 Institut für Softwaretechnik und Interaktive Systeme, 2012. Cited on page 36.

[177] J. Schönböck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer, *Catch me if you can – debugging support for model transformations*, in Proceedings of the 2009 International Conference on Models in Software Engineering, MODELS'09, Berlin, Heidelberg, 2010, Springer-Verlag, pp. 5–20. Cited on page 36.

[178] J. Schönböck, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger, and W. Schwinger, *Debugging model-to-model transformations*, in 2012 19th Asia-Pacific Software Engineering Conference, vol. 1, Dec 2012, pp. 164–173. Cited on page 36.

[179] A. Schürr, *Specification of graph translators with triple graph grammars*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 151–163. Cited on pages 11 and 157.

[180] B. Selic, *The pragmatics of model-driven development*, Software, IEEE, 20 (2003), pp. 19–25. Cited on pages 1 and 8.

[181] S. Sendall and W. Kozaczynski, *Model transformation: The heart and soul of model-driven software development*, IEEE Software, 20 (2003), pp. 42–45. Cited on page 11.

[182] B. Siegmund, M. Perscheid, M. Taeumel, and R. Hirschfeld, *Studying the advancement in debugging practice of professional software developers*, in Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on, Nov 2014, pp. 269–274. Cited on page 30.

[183] J. Silva, *A survey on algorithmic debugging strategies*, Advances in Engineering Software, 42 (2011), pp. 976 – 991. Cited on page 30.

[184] S. Singh and L. Singh, *Study of current program slicing techniques*, in Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference -, Sept 2014, pp. 810–814. Cited on page 29.

[185] M. Sjölund, F. Casella, A. Pop, A. Asghar, P. Fritzson, W. Braun, L. Ochel, B. Bachmann, and P. Milano, *Integrated Debugging of Equation-Based Models*, in Proceedings of the 10th International ModelicaConference, 2014, pp. 195–204. Cited on page 35.

[186] M. Sjölund and P. Fritzson, *Debugging Symbolic Transformations in Equation Systems*, in EOOLT, Linköping Electronic Conference Proceedings vol. 56, 2011, pp. 67–74. Cited on pages 35 and 164.

[187] D. Stewart and M. M. Chakravarty, *Dynamic applications from the ground up*, in Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, 2005, pp. 27–38. Cited on page 34.

[188] L. Stockmann, *Debugging models in the context of automotive software development*, in Proceedings of the Doctoral Symposium of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, M. Chechik and D. Kolovos, eds., 29 Sept. 2015. Cited on page 34.

[189] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, USA, 1977. Cited on page 14.

[190] Y. Sun and J. Gray, *End-user support for debugging demonstration-based model transformation execution*, in Modelling Foundations and Applications, P. Van Gorp, T. Ritter, and L. Rose, eds., vol. 7949 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 86–100. Cited on page 36.

[191] E. Syriani and H. Vangheluwe, *A modular timed graph transformation language for simulation-based design*, Software & Systems Modeling, 12 (2013), pp. 387–414. Cited on page 11.

[192] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, *AToMPM: A web-based modeling environment*, in Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), vol. 1115, CEUR, Sept. 2013, pp. 21–25. Cited on pages 10 and 61.

[193] S. L. Tanimoto, *VIVA: A visual language for image processing*, Journal of Visual Languages and Computing, 1 (1990), pp. 127–139. Cited on page 34.

[194] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson, *Replay Debugging of Real-Time Systems Using Time Machines*, in Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), IEEE Computer Society, 2003, pp. 288–295. Cited on page 33.

[195] J. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi, *A noninterference monitoring and replay mechanism for real-time software testing and debugging*, Software Engineering, IEEE Transactions on, 16 (1990), pp. 897–916. Cited on page 33.

[196] A. M. Uhrmacher, *Dynamic structures in modeling and simulation: A reflective approach*, ACM Trans. Model. Comput. Simul., 11 (2001), pp. 206–232. Cited on page 122.

[197] Z. Ujhelyi, A. Horvath, and D. Varro, *Towards dynamic backward slicing of model transformations*, in Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, Washington, DC, USA, 2011, IEEE Computer Society, pp. 404–407. Cited on page 36.

[198] D. Ungar and R. B. Smith, *Self: The power of simplicity*, SIGPLAN Not., 22 (1987), pp. 227–242. Cited on page 34.

[199] M. G. J. van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju, *TIDE: A generic debugging framework — tool demonstration —*, Electron. Notes Theor. Comput. Sci., 141 (2005), pp. 161–165. Cited on page 38.

[200] T. van der Storm, *Semantic deltas for live DSL environments*, in Proceedings of the 1st International Workshop on Live Programming, LIVE '13, Piscataway, NJ, USA, 2013, IEEE Press, pp. 35–38. Cited on pages 38 and 176.

[201] Y. Van Tendeloo, *Foundations of a multi-paradigm modelling tool*, in Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015, 2015. Cited on page 67.

[202] Y. Van Tendeloo, B. Barroca, S. Van Mierlo, and H. Vangheluwe,

Modelverse specification, tech. report, University of Antwerp, 2016. Cited on page 69.

[203] Y. Van Tendeloo, S. Van Mierlo, B. Meyers, and H. Vangheluwe, *Concrete syntax: A multi-paradigm modelling approach*, in Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, New York, NY, USA, 2017, ACM, pp. 182–193. Cited on page 9.

[204] Y. Van Tendeloo and H. Vangheluwe, *The modular architecture of the Python(P)DEVS simulation kernel*, in TMS-DEVS, SpringSim, 2014, pp. 387–392. Cited on page 126.

[205] Y. Van Tendeloo and H. Vangheluwe, *PythonPDEVS: a distributed Parallel DEVS simulator*, in Proceedings of the 2015 Spring Simulation Multiconference, SpringSim '15, Society for Computer Simulation International, 2015, pp. 844–851. Cited on page 94.

[206] Y. Van Tendeloo and H. Vangheluwe, *An overview of PythonPDEVS*, in JDF 2016, 2016, pp. 59–66. Cited on page 126.

[207] Y. Van Tendeloo and H. Vangheluwe, *The Modelverse: a tool for multi-paradigm modelling and simulation*, in Proceedings of the 2017 Winter Simulation Conference, WSC 2017, IEEE, Dec. 2017, pp. 944 – 955. Cited on pages 69 and 185.

[208] H. Vangheluwe, *Foundations of modelling and simulation of complex systems*, ECEASST, 10 (2008). Cited on page 8.

[209] H. Vangheluwe and G. Vansteenkiste, *A multi-paradigm modeling and simulation methodology: formalisms and languages*, in Proceedings of the 1996 European Simulation Symposium (Genoa), Society for Computer Simulation International, 1996, pp. 168–172. Cited on page 16.

[210] G. Wainer and N. Giambiasi, *Timed Cell-DEVS: Modeling and Simulation of Cell Spaces*, Springer New York, New York, NY, 2001, pp. 187–214. Cited on pages 49 and 164.

[211] P. Wang, X. Zhang, P. Hao, and Y. Zhang, *Towards the multithreaded deterministic replay in program debugging*, in Information Science and Digital Content Technology (ICIDT), 2012 8th International Conference on, vol. 1, June 2012, pp. 139–144. Cited on page 32.

[212] R. Wille, M. Soeken, and R. Drechsler, *Debugging of inconsistent UML/OCL models*, in 2012 Design, Automation Test in Europe Conference Exhibition (DATE), March 2012, pp. 1078–1083. Cited on page 38.

[213] M. Wimmer, A. Kusel, J. Schoenboeck, G. Kappel, W. Retschitzegger, and W. Schwinger, *Reviving QVT relations: Model-based debugging using colored Petri nets*, in Model Driven Engineering Languages and Systems, A. Schürr and B. Selic, eds., vol. 5795 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 727–732. Cited on page 36.

[214] S. Wolfram, *Cellular automata as models of complexity*, Nature, 311 (1984), pp. 419 EP –. Cited on pages 49 and 164.

[215] W. E. Wong and V. Debroy, *A survey of software fault localization*, tech. report, University of Texas, Dallas, 2009. Cited on page 29.

[216] H. Wu, J. Gray, and M. Mernik, *Grammar-driven generation of domain-specific language debuggers*, Software: Practice and Experience, 38 (2008), pp. 1073–1103. Cited on pages 37 and 164.

[217] Z. Xing, J. Sun, Y. Liu, and J. S. Dong, *SpecDiff: Debugging formal specifications*, in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, New York, NY, USA, 2010, ACM, pp. 353–354. Cited on page 36.

[218] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, *Supporting automatic model inconsistency fixing*, in Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, New York, NY, USA, 2009, ACM, pp. 315–324. Cited on page 38.

[219] F. Zalila, X. Crégut, and M. Pantel, *A transformation-driven approach to automate feedback verification results*, in Proceedings of the Third International Conference on Model and Data Engineering - Volume 8216, MEDI 2013, New York, NY, USA, 2013, Springer-Verlag New York, Inc., pp. 266–277. Cited on page 40.

[220] C. Zamfir and G. Candea, *Execution synthesis: A technique for automated software debugging*, in Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, New York, NY, USA, 2010, ACM, pp. 321–334. Cited on page 32.

[221] B. P. Zeigler, *Theory of Modelling and Simulation*, Krieger Publishing Co., Inc., Melbourne, FL, USA, 1984. Cited on pages 2, 8, 13, 36, 43, and 88.

[222] M. V. Zelkowitz, *Reversible execution*, Commun. ACM, 16 (1973), pp. 566–566. Cited on page 34.

[223] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. Cited on pages 2, 3, and 28.

[224] A. Zeller and D. Lütkehaus, *DDD - a free graphical front-end for UNIX debuggers*, SIGPLAN Not., 31 (1996), pp. 22–27. Cited on page 29.

[225] Y. Zhang and B. Xu, *A survey of semantic description frameworks for programming languages*, SIGPLAN Notices, 39 (2004), pp. 14–30. Cited on page 13.

[226] K. Zurowska and J. Dingel, *Language-specific model checking of UML-RT models*, Software & Systems Modeling, 16 (2017), pp. 393–415. Cited on page 2.