Formal Semantic Specification of Domain-Specific Modeling Languages for Cyber-Physical Systems

By

Gabor Simko

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

August, 2014

Nashville, Tennessee

Approved:

Professor Janos Sztipanovits

Professor Gabor Karsai

Professor Xenofon Koutsoukos

Professor Jeremy Spinrad

Ethan Jackson, Ph.D.

Professor Sandeep Neema

DEDICATION

To my family, whose support, encouragement and love have sustained me throughout my life.

ACKNOWLEDGMENTS

First and foremost I am grateful to my advisor Janos Sztipanovits for his support and guidance during my academic years. It was his encouragement and visionary ideas that helped me through the hard times, and which finally led to this dissertation.

I would also like to thank Ethan Jackson for his continuous support and faith in me, and for the opportunities to work with him at Microsoft Research. I am very much indebted for all his help in building my career and for teaching many invaluable lessons.

I am grateful for all the committee members: Sandeep Neema, Gabor Karsai, Xenofon Koutsoukos and Jeremy Spinrad for their time, interest and comments.

I would also like to thank Tihamer Levendovszky for his many help on my journey to become a researcher. He greatly improved my academic writing skills, and whenever I got stuck in research, he always had some great advices.

My time at Vanderbilt University was made enjoyable in large part to my colleagues and friends. In particular, I express a deep appreciation to Csanad Szabo, Tamas Szarka, Janos Mathe, Tamas Kecskes, David Lindecker, Heath LeBlanc, Debargha Dey, Zhenkai Zhang, Emeka Eyisi, Mark Yampolskiy, Siyuan Dai, Peter Horvath, Janos Sallai, Zsolt Lattmann, Benjamin Babjak, Joe Porter, Daniel Balasubramanian and Amin Ghafouri.

I am also thankful to Grit Denker, Natarajan Shankar and Ashish Tiwari for their support during my time at SRI International. In a distant collaboration, it was a great pleasure to work with Pieter Mosterman.

I would like to thank my family for all their love, support and encouragement, despite the difficulties of living so far away from each other. My parents, who raised me with the love of science and who did everything to support me on my journey. My brother Benedek Simko, for our conversations about information technology, for his faith in me, and for his guidance since our childhood. I would also like to thank Nora Balint for her loving, faithful and encouraging support throughout these years. Thank you!

PREFACE

Model-Integrated Computing is increasingly used for designing Cyber-Physical Systems (CPS), since it increases productivity and product quality through simulators, automated testing, code generators and verification tools. In this approach, models are represented using Domain-Specific Modeling Languages (DSMLs). A DSML is defined by its syntax and semantics, and while meta-modeling (and meta-modeling environments) provides a mature methodology for tackling the syntax of DSMLs, expressing the semantics of a DSML is still in its infancy. Without unambiguous specifications, different tools may interpret the languages in different ways, which could easily lead to situations when the compiler generates code with different behavior than what the verification tool analyzes. Therefore, in order to help the development of consistent tools, we need to formalize the semantics of these languages.

In this work, we discuss the formalization of the structural and behavioral semantics of CPS DSMLs using a logic programming based approach. We introduce ForSpec, an executable formal specification language for the structural and behavioral semantics of CPS DSMLs. ForSpec is a constraint logic programming language based on fixed-point logic over algebraic data types with support for both denotational and operational specifications.

In order to help the development of denotational semantic specifications, we introduce an extension of the semantic anchoring framework, and define several reusable semantic units for CPS modeling languages in ForSpec. Using these semantic units, we demonstrate the complete formalization of the structural and denotational semantics of a bond graph language and a CPS modeling language.

Finally, in order to demonstrate operational specifications in ForSpec, we develop the structural and operational semantic specifications for the MathWorks Stateflow language.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

PREFACE iv

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF ABBREVIATIONS x

Chapter

1. Introduction 1

2. Challenges in model-based design of Cyber-Physical Systems 6

   2.1. Model-based design for Cyber-Physical Systems 6
   2.2. Causality 7
   2.3. Semantic domains for time 8
   2.4. Interaction models for computational processes 10
   2.5. Conclusion 11

3. Semantics of CPS domain-specific modeling languages 12

   3.1. Syntax and semantics 12
   3.2. Structural semantics 15
   3.3. Behavioral semantics 16
      3.3.1. Denotational semantics 16
      3.3.2. Translational semantics 20
      3.3.3. Operational semantics 20
   3.4. Conclusion 25

4. Formal specification languages for CPS DSMLs 27

   4.1. Specification approaches 28
      4.1.1. Weaving approach 28

4.1.2. Rewriting approach . . . 29
4.1.3. Translational approach . . . 30
4.2. Specification languages . . . 30
4.2.1. Constraint logic programming languages . . . 30
4.2.2. Rewriting logic . . . 32
4.2.3. Abstract State Machine languages . . . 34
4.2.4. High-level specification languages . . . 36
4.3. Conclusion . . . 37

5. A formal language for semantic specifications . . . 40

5.1. Introduction to ForSpec . . . 40
5.1.1. Core language . . . 40
5.1.2. Full language . . . 44
5.2. Semantic specification in ForSpec/Formula . . . 51
5.2.1. Operational semantics in ForSpec . . . 51
5.2.2. Translational and denotational semantics . . . 54
5.3. Conclusion . . . 57

6. Reusable semantic units for formalizing the denotational semantics of CPS modeling languages . . . 59

6.1. Semantic anchoring . . . 59
6.2. Primary CPS semantic units . . . 61
6.2.1. Differential algebraic equations . . . 61
6.2.2. Difference equations . . . 64
6.2.3. Finite-state machines . . . 66
6.3. Semantic unit extension and restriction . . . 66
6.3.1. Ordinary differential equations . . . 67
6.3.2. Linear ordinary differential equations . . . 68
6.3.3. Tagged differential algebraic equations . . . 69
6.3.4. Parallel hybrid automata . . . 69
6.4. Composition of semantic units . . . 72
6.4.1. Hybrid differential-difference equations . . . 72
6.4.2. Hybrid automata . . . 73
6.5. Conclusion . . . 75

7. Case studies . . . 76

7.1. Specification of a bond graph language . . . 76
7.1.1. Bond graph structural semantics . . . 77

7.1.2. Bond graph denotational semantics 82
7.1.3. Example 85
7.1.4. Extensibility 88
7.1.5. Conclusion 92
7.2. Specification of the Cyber-Physical Systems Modeling Language 93
7.2.1. Formalization of semantics 95
7.2.2. Formalization of language integration 101
7.2.3. Conclusion 110
7.3. Specification of the Stateflow language 111
7.3.1. Introduction 111
7.3.2. Related work 112
7.3.3. Semantics by example 113
7.3.4. Informal semantics 115
7.3.5. Structural semantics 120
7.3.6. Operational semantics 127
7.3.7. Testing 133
7.3.8. Conclusion 134

8. Conclusion 136

REFERENCES 138

LIST OF TABLES

Table | Page

1. Comparison of different specification languages for Domain-Specific Modeling Language (DSML) specification. 39

2. Use cases for the ForSpec specifications. 58

LIST OF FIGURES

Figure Page

1. Formalization of behavioral semantics by semantic anchoring. The behavioral semantics of language $\mathcal{L}$ is expressed as a semantic anchoring M_S from its abstract syntax $A_{\mathcal{L}}$ to the abstract syntax A_{SU} of the semantic unit. The semantic mapping M_{S_i} and the mathematical domain are unique for the semantic unit, and developed only once. 60

2. Examples for restriction, extension and composition of semantic units. . . . 67

3. A quarter car suspension model using the bond graph language. 77

4. A language for bond graphs as defined in Generic Modeling Environment (GME). 78

5. Bond graph model for an RLC series circuit. 86

6. A hybrid bond graph language. 90

7. A hybrid bond graph model for a lamp with a switch. 90

8. GME meta-model for the composition sub-language of CyPhyML 93

9. Denotational semantic specification of CyPhyML using semantic anchoring for the semantic interface of the integrated languages and the integration language itself. 98

10. A tricky Stateflow example. 113

11. Testing harness for the specifications. 134

LIST OF ABBREVIATIONS

ASM Abstract State Machine. 28, 29, 34, 35, 37

CPS Cyber-Physical System. 1–8, 11, 26, 31, 38, 39, 58, 72–74, 91, 107, 134, 135

CyPhyML Cyber-Physical Systems Modeling Language. 91, 95, 99, 100

DAE Differential Algebraic Equation. 59, 60, 62, 65, 67, 68, 70–72, 75, 80, 90

DE Difference Equation. 62

DSL Domain-Specific Language. 29, 33

DSML Domain-Specific Modeling Language. vii, 1, 2, 5–7, 13, 15, 28–39, 56, 80, 134

DSVL Domain-Specific Visual Language. 34

ESMoL Embedded Systems Modeling Language. 54, 99–101

FSM Finite-State Machine. 28, 29, 35, 43, 50, 59, 63, 64, 71

GME Generic Modeling Environment. viii, 6, 76, 77, 79, 87

LTL Linear Temporal Logic. 33, 38

LTS Labeled Transition System. 11

MEL Membership Equational Logic. 33, 37

MIC Model-Integrated Computing. 1

MoC Model of Computation. 10

MOF MetaObject Facility. 33

MSOS Modular Structural Operational Semantics. 23, 25

OCL Object Constraint Language. 32, 34, 77, 79

ODE Ordinary Differential Equation. 65, 66

OMG Object Management Group. 29

QVT Query/View/Transformation. 29

SAT Satisfiability. 31

SMT Satisfiability Modulo Theories. 4, 30, 135

SOS Structural Operational Semantics. 21–25, 125

TS Transition System. 21

UML Unified Modeling Language. 32, 36, 37, 76

CHAPTER 1

INTRODUCTION

Cyber-Physical Systems (CPSs) are integration of computational and physical systems. CPSs have gained significant traction in the past few decades with major applications in the automotive industry, avionics, health-care systems, plants, smart grids and manufacturing.

Recently, Model-Integrated Computing (MIC) has become a well-established discipline for designing complex CPS systems [63][125]. In this approach, both computational and physical systems are modeled using computer-based tools, and the models are used for performing various activities, such as running virtual tests (computer simulations), generating software implementations, checking for safety properties with formal analysis and formal verification tools and others. Such an approach mitigates the risks of faulty system designs by pinpointing possible design flaws during the design phase, and by leveraging automated code generation instead of error-prone manual coding. In summary, MIC improves product quality, while decreases development time and costs by the usage of automated tool-suites.

In the MIC approach, models are represented using Domain-Specific Modeling Languages (DSMLs). These are relatively small languages tailored for describing the concepts of specific domains. As any language, a DSML is defined by its syntax and semantics: the syntax describes the structure of the language (e.g., syntactic elements and their relations), and the semantics describes the meaning of its models.

While meta-modeling (and meta-modeling environments) provides a mature methodology for tackling the syntax of DSMLs, expressing the semantics of DSMLs is still in its infancy. Nonetheless, the semantics of a language should not be taken lightly, especially not in the CPS domain. Without unambiguous specifications, different

tools may interpret the language in different ways, which could easily lead to situations when the compiler generates code with different behavior than what the verification tool analyzes. This can potentially render the results of the formal analysis and verification tools invalid. Furthermore, developing the formal semantics of a language requires the developer to clearly think over all the details of the language, therefore avoiding common design errors in the language. *In order to support the development of a CPS DSML along with tools operating on the language, it is highly recommended to rigorously define and formalize both the syntax and semantics of the language*. Of course, even with formal semantic specifications, a tool may be faulty with respect to these specifications, but as long as the specifications are unambiguous, this is a problem with the tool, and not a problem with the language.

In this paper, we develop a framework for specifying the semantics of typical CPS DSMLs. As most of the research is found in the semantics of programming languages, first, let us enumerate some of the differences between CPS modeling languages and regular programming languages, in order to demonstrate some of the challenges we have to circumvent (some of them already discussed in [57]):

- **Syntax:** while the syntax of programming languages are usually described using grammars, and their abstract structure is an abstract syntax tree, the syntax of modeling languages is more complicated. The syntax of DSMLs is usually described using meta-models, and the underlying abstract syntax is an abstract syntax graph.
- **Structure:** while programming languages have static semantics, the analogous *structural semantics* of modeling languages is not necessarily static. For example, models and model transformations can describe dynamically changing structures.
- **Behavior:** in programming languages time is completely abstracted away. In contrast, the notion of time is essential for CPS models. Furthermore, while programs always describe sequences of computations, CPS models can describe other behaviors as well (e.g., trajectories over continuous time and space).

In order to specify the semantics of a language, we need a specification language. Ideally, we need a specification language satisfying the following requirements:

- **Mathematical foundations:** the specification language needs to be unambiguous, and have a rigorous mathematical foundation to support sound reasoning.
- **Comprehensibility:** in order to help the human reader, the language should be lightweight and easy to understand. The language should provide a good balance between expressiveness and comprehensibility.
- **Executability:** the language should be executable to help debugging the specifications, as well as facilitate rapid prototyping of compilers and interpreters.
- **Support for meta-modeling:** The language should provide structures for expressing meta-models and models with ease; preferably in their abstract syntax graph form (in contrast with a tree representation, where cross-references are resolved by identifiers).
- **Versatility:** the language should be able to define behaviors in different styles, such as operational or denotational semantics, as well as to describe the structural semantics.
- **Symbolic representation:** the language should be able to operate on a symbolic level in order to support advanced model verification techniques, such as symbolic model checking or model finding.

FORMULA – a constraint logic programming language developed at Microsoft Research – performs well with regards to these requirements, and FORMULA was already used for the structural semantic specifications in [57]. In this work, we present an extension of the FORMULA language to support the specification of the behavioral semantics of CPS modeling languages. This has the advantage that both the structural and behavioral semantics are specified using the same formalism, and therefore they can be used together for formal analysis. Other advantages of our approach are the following:

- **Comprehensibility.** We provide a "literate programming" environment, where formal specifications and informal documentation are written within the same document, therefore helping comprehensibility of the specifications. The same specifications are used for execution, as well as for generating high-quality (formatted) documentation for the language. This helps keeping the documentation and specifications consistent.

- **Unified specifications.** Both the structural and behavioral semantics are described using the same formal language, thus both of them are available for the verification tool(s). Furthermore, this is a significant advantage for users, as they need to learn only one language.

- **Executable specifications.** The specifications are executable, therefore they can be used for prototyping languages: the structural semantics can be used for model conformance checking, the operational semantics for generating execution traces, and the denotational semantics for automated compilation and code generation.

- **Symbolic evaluation.** The specifications can be compiled to symbolic expressions that can be solved by state of the art Satisfiability Modulo Theories (SMT) solvers. This can be leveraged for automated test-case generation, bounded model checking, design-space exploration and resource allocation, such as scheduling, deployment and others.

Our contributions are the following:

1. ForSpec, a language based on FORMULA, that provides improved support for behavioral semantic specifications.

2. Identification of different semantic specification styles in FORMULA/ForSpec.

3. Specification of reusable semantic units, which are used by subsequent behavioral semantic specifications.

4. Semantic specification of CPS languages, in particular, the most complete formal specification of the MathWorks Stateflow language at present.

5. Tool support for the specifications: a compiler for ForSpec, and a LaTeX documentation generator that facilitates "literate programming" in ForSpec.

The paper is organized as follows. Chapter 2 describes the challenges related to the model-based design of CPS. These challenges are related to the heterogeneous behavior of CPS, and therefore directly affect the behavioral semantic specifications. In Chapter 3, modeling languages and their semantics are discussed. Chapter 4 is a review of existing approaches and languages for formal semantic specification of DSMLs. Chapter 5 introduces a formal specification language for CPS modeling languages. In Chapter 6, we discuss the extension of the semantic anchoring framework to denotational specifications, and describe several reusable semantic units. We use these semantic units to develop several case studies in Chapter 7. Finally, in Chapter 8 we conclude with the contributions and possible future work.

CHAPTER 2

CHALLENGES IN MODEL-BASED DESIGN OF CYBER-PHYSICAL SYSTEMS

2.1 Model-based design for Cyber-Physical Systems

Cyber-Physical Systems (CPSs) are integration of computational and physical systems. One of the key challenges in the engineering of CPSs is the integration of heterogeneous concepts, tools and languages [125]. In order to address these challenges, a model-integrated development approach for CPS design is introduced by Karsai and Sztipanovits [63], which advocates the pervasive use of models throughout the design process: such as application models, platform models, physical system models, environment models, and the interaction models between these models. For embedded systems, a similar approach is discussed in [124][65], in which both the computational processes as well as the supporting architecture (hardware platform, physical architecture, operating environment) are modeled within a common modeling framework.

Arguably, modeling the vastly diverse set of CPS concepts using a single generic modeling language is impractical. Instead, we can adopt a set of Domain Specific Modeling Languages (DSMLs) – each of which describes a particular domain –, and interconnect them through a model-integration language. This is well aligned with the content creation as well: models are created by domain experts, who are used to their own domain-specific concepts and terminology. In order to help their work, we need to provide an environment (such as the Generic Modeling Environment (GME) [68]) that allows them to work with these domain-specific languages. In the CPS domain, typical examples for Domain-Specific Modeling Languages (DSMLs) are circuit diagrams for electrical engineers, or data-flow diagrams for control engineers.

An important question in CPS modeling is how to model the integration of the individual domains, such that the cross-domain interactions are properly accounted for

[125]. A recently proposed solution [134][67][117] uses a model-integration DSML for describing model interactions. Such a model-integration language is built upon the paradigm of component-based modeling, where complex systems are built from interconnected components. From the model integration point of view, the most important part of a component is its interface: through which it interacts with its environment. Then, the role of the model-integration language is to unambiguously define the heterogeneous interactions (e.g., message passing, variable sharing, function call or physical interactions) between the interconnected interfaces.

As discussed in the introduction, for CPS modeling, we need unambiguous semantic specifications for the modeling languages. One of the key challenges for developing these specifications is found in the heterogeneity of the behaviors represented by the languages: ranging from untimed discrete computations to trajectories over continuous time and space.

In CPS modeling, we can distinguish four fundamental dimensions of heterogeneity that affects the semantics of the languages:

- causality considerations (causal vs. acausal relations),
- time semantics (e.g. continuous-time, discrete-time, etc.),
- physical domains (e.g. electrical, mechanical, thermal, acoustic, hydraulic),
- interaction models (e.g. MoC-based, interaction algebra).

2.2 Causality

Classical system theory and control theory are traditionally based on input/output signal flows, where causality – that the inputs of a system determine its outputs – plays a key role. However, such a causal model is artificial and inapplicable for physical systems modeling [132], because the separation of inputs and outputs is generally unknown at the time of modeling. The problem stems from the mathematical models of physical laws and systems: these models are equation based and there are no causal relationships between the involved variables. Indeed, the only causal

law in physics is the second law of thermodynamics [11], which defines the direction in which the time flows. Recently, acausal physical modeling has gained traction and several acausal physical systems modeling languages have been designed, such as the bond graph formalism [62], Modelica [35, 8], Simscape [76], EcosimPro [28] and others.

Even though the mathematical models for the laws of physics are acausal, we often rely on their causal abstraction: for example, operational amplifiers are often abstracted as input-output systems, such that the output is solely determined by its input and it has no feedback effects on the input. For real operational amplifiers these assumptions are only approximations, but they are reasonably accurate in many situations and they greatly simplify the design process. Therefore, physical modeling languages usually support both causal and acausal modeling. Clearly, this is the case with all state of the art modelers and languages (e.g. Simulink/Simscape, Modelica, Bond graph modeling with modulated elements [62], etc.). Consequently, our behavioral semantic specifications need to be able to describe both acausal and causal models.

2.3 Semantic domains for time

In software design, one of the most powerful abstraction is the abstraction of time [69]. Such an abstraction is harmless as long as time is a non-functional property of programs. However, in CPSs and real-time systems this premise is often invalid [70], and timing is a key concept from a functional point of view of the system. For example, in hard real time systems, results are often worthless or – worst-case – even catastrophic, if not delivered in time.

Therefore, in CPSs, we have to properly account for the timely behavior of models. It is known that the behaviors of CPS models are interpreted over a diverse set of semantic domains for time. Let us summarize some of these semantic domains in the following.

Logical time is a time model for computational processes, whose behaviors are described by sequences of states. Logical time is a countable totally ordered time domain, and is typically represented by the natural numbers.

Continuous-time (also known as real time, physical time) is a dense time model and is often represented by the non-negative real numbers (where zero represents the start of the system) or the non-negative hyper-reals. Physical systems are usually considered to have their behavior in the continuous-time regime.

Super-dense time [74] is a time model that extends real time with causal ordering for simultaneous events. Super-dense time is usually used for describing the behavior of discrete-event systems [71].

Discrete-time denotes a finite number of events on any finite time interval. Therefore, the events can be indexed by the natural numbers. For instance, clock-driven systems have discrete-time semantics.

Hyper-dense time [88, 89] is a time model that extends hyper-real time with causal ordering for simultaneous events. Hyper-dense time is used for describing discontinuities in the behavior of physical systems.

In real-time systems, time is a functional property and we need to explicitly account for the execution times of the software. However, on modern architectures, the exact execution times cannot be accurately predicted because of caching, pipelining and other advanced techniques. In order to resolve the problem, several abstractions of time were proposed [66], such as Zero Execution Time, Bounded Execution Time and Logical Execution Time.

The *Zero Execution Time* (ZET) model abstracts away time by assuming that the execution time is zero, in other words, the computation is infinitely fast. ZET is the basis for synchronous reactive programming.

In the *Bounded Execution Time* (BET) model execution time has an upper bound. A program execution is correct as long as the output is produced within this temporal bound. Note that this model is not really an abstraction, rather a specification that can be verified for correctness.

The *Logical Execution Time* [50] (LET) model abstracts away the real execution time but does not discard it completely as ZET. LET denotes the time it takes from reading the input to producing the output, regardless of the real execution time. Compared to BET, the LET abstraction also defines a lower bound equal to the upper bound, i.e., LET precisely defines the time when the output is produced. Such an abstraction has significant impact on the complexity of the behavior, for example, it protects against timing anomalies [102] (i.e., when a faster local execution leads to slower global execution).

Timed automata are introduced as a time model for real-time systems by Abdellatif et al. [1]. Compared to LET semantics, timed automata provide more generic constraints, such as lower time bounds, upper time bounds and time non-determinism. To avoid timing anomalies, the authors introduce the notion of time-robustness along with some sufficient conditions to guarantee it.

2.4 Interaction models for computational processes

When modeling heterogeneous computational systems, a key question is how to define the interactions between different subsystems. We can distinguish two approaches here.

The interactions may be modeled with Model of Computations (MoCs), e.g., in Ptolemy II [37, 38]. Ptolemy II is based on the hierarchical composition of various MoCs, such as process networks, dynamic and synchronous dataflows, discrete-event systems, synchronous-reactive systems, finite-state machines and modal models. Each actor in the hierarchy has a director – a Model of Computation – that determines the interaction model between its children. The *actor abstract semantics* [71] of Ptolemy II is a common abstraction of all MoCs in Ptolemy II, which defines an interface with action methods and method contracts. An actor that conforms to these contracts is said to be domain polymorphic, and a domain polymorphic actor is compatible with any director.

Alternatively, the interactions may be modeled using algebras, e.g., in the BIP [13] framework. In BIP, connector and interaction algebras [15] are defined for

characterizing different interaction types: such as rendezvous, broadcast, or atomic broadcast. In the BIP framework, the behaviors of components are abstracted as Labeled Transition Systems (LTSs) and the interaction algebra establishes the relations between these transition systems.

2.5 Conclusion

In this chapter, we discussed some of the challenges in the modeling of heterogeneous CPSs, such as causality, timing and different interaction models. These concerns have important consequences on semantic specifications: due to them, the classical results of programming language specifications are only applicable to the computational languages of CPSs. In order to express the semantics of other CPS languages, such as physical modeling languages and model-integration languages, we need to extend the existing specification styles as proposed in the rest of this thesis.

CHAPTER 3

SEMANTICS OF CPS DOMAIN-SPECIFIC MODELING LANGUAGES

In this chapter, we discuss the syntax and semantics of modeling languages and existing work in semantic specifications. In particular, we discuss the structural, denotational, translational and operational semantics of languages.

3.1 Syntax and semantics

In [21], a (modeling) language is defined as a tuple of $\langle C, A, S, M_A, M_S \rangle$, the concrete syntax C, the abstract syntax A, the semantic domain S, and two mappings $M_A \colon C \to A$ and $M_S \colon A \to S$ that map from the concrete syntax to the abstract syntax, and from the abstract syntax to the semantic domain, respectively.

The concrete syntax of a language is the concrete representation that is used for representing programs/models. Traditional programming languages are usually text-based, whereas modeling languages often have visual representations. A language describes a set of concepts and relations between these concepts, which is represented by its abstract syntax. The syntactic mapping M_A maps elements of the concrete syntax to corresponding elements of the abstract syntax. Finally, semantics defines the meaning of models by means of semantic mapping(s) M_S from the concepts and relations of the abstract syntax to some semantic domain(s).

In order to discuss existing work in language semantics, first, we need to understand the syntax and semantics of programming languages. The concrete syntax of a programming language is usually described with some context-free grammar – typically in the (extended) Backus-Naur form (BNF) [9]. Such a grammar describes production rules, which can be used to build parse trees from the source code. The abstract syntax tree of a program is an abstract version of the parse tree, which typically removes some parser-specific details from the parse tree. The *static semantics* of a

programming language describes those properties of its programs that are statically (that is, without executing the program) computable; this is also called the well-formedness rules of the language. Typically, this corresponds to context-sensitive parts of the grammar that are not expressible with a context-free grammar: such as unique naming of variables, static type checking and scoping. Finally, the *dynamic semantics* of a language describes its dynamic aspects: what are the sequences of computations described by its programs.

In contrast, the syntax of a Domain-Specific Modeling Language (DSML) is typically described with a meta-model. Generally, meta-models describe graph structures, thus models are represented using abstract syntax graphs. Furthermore, instead of static semantics, models have *structural semantics* [57]. In [22], structural semantics is defined as the meaning of the models in terms of the structure of model instances. Similar to static semantics, structural semantics describes the well-formedness rules of a language; however, structural semantics is not necessarily static. In model-based design, models may represent dynamic structures that evolve through model transformations, in which case the structural semantics describes invariants for these transformations. The dynamic behavior of a model is described by its *behavioral semantics*. Note that the behaviors represented by modeling languages are generally interpreted on different semantic domains than those of programming languages. For example, models can represent physical systems, for which the behaviors are trajectories over the continuous time and space.

In the following, we discuss some of the most well-known specification styles for expressing the behavior of (programming) languages. Note that the list is far from being complete – other approaches are discussed in [85].

One of the earliest styles for semantic specifications is the *denotational semantics* – formerly known as mathematical semantics – introduced by Christopher Strachey and Dana Scott in the late 1960s [113, 114, 115, 122]. Denotational semantics describes the language semantics by mapping its phrases to mathematical objects, such as numbers, tuples, functions, etc. An advantage of denotational semantics is that it provides mathematically rigorous specifications for programs without specifying any

computational procedures for calculating the results. This results in abstract specifications that describe *what* programs do instead of describing *how* they do it. Such an approach is especially useful for reasoning about equivalence of programs [120].

Operational semantics (the term possibly coined by Dana Scott [113]), in contrast, defines a procedure, which – if executed – results in the semantics of the program. The most well-known approaches to operational semantics are the Structural Operational Semantics [97, 61] and Abstract State Machines [40]. An advantage of operational semantics is that the specifications are natural and relatively easy to understand in general. A disadvantage is that understanding the consequences of the specifications is challenging.

Algebraic semantics [30] uses algebraic structures and equational logic to describe the semantics of programs. The idea is to define modules that consist of many-sorted signatures Σ and equality axioms E. Then, a corresponding algebraic structure (that is, an interpretation for the signature: a carrier set and functions) is an implementation of the module, if the equalities hold in the structure. This is particularly efficient for specifying abstract data types, since the specification only describes the properties of the operations. Furthermore, algebraic semantics can efficiently describe algebraic data types and programs [120]. In this approach, the program is represented using the term algebra T_Σ over Σ. From T_Σ, a quotient algebra $T_{\Sigma/E}$ (isomorphic to any initial algebra) is obtained by factoring out the congruences described by equations E. Then, the semantics of the program is the initial algebra (selecting one from the isomorphic initial algebras).

Axiomatic semantics was introduced by Robert Floyd [34] and C.A.R. Hoare [51] in the late 1960s. Axiomatic semantics is even more abstract than denotational semantics, as there is no notion for the state of a machine. Instead, the semantics is established by pre- and postconditions described in predicate logic.

In summary, operational semantics defines a computational process transforming states of a machine, denotational semantics defines a mathematical function that describes state transformations, algebraic semantics defines programs by describing congruences, and axiomatic semantics defines programs by establishing predicates over their behavior.

3.2 Structural semantics

Structural semantics describes the well-formed structures of a modeling language, which is defined by a set of constraints that express the well-formedness rules of the language. A model is said to conform to its meta-model, if it satisfies all these constraints.

While analogous to static semantics, there are a number of differences between static and structural semantics. First, since the abstract syntaxes of DSMLs are generally graph structures, some well-formedness rules are unique for the structural semantics: for instance, the lack of cycles [54] in a graph. Second, structural semantics can describe dynamic properties. For example, given a graph model for the network topology of some communicating agents, the structural semantics can describe constraints over the connectivity of the graph during its dynamic reconfigurations. Third, for some languages (e.g., UML class diagram) structural semantics is the only semantics, because they do not represent behaviors.

Jackson et al. [57, 60] introduce a formal model for defining the structural semantics of DSMLs in a logic framework. In this work, a DSML $L = \langle D, (\llbracket\rrbracket_j)_{j \in J} \rangle$ is a pair comprised of a domain D and a set of interpretations.

The domain $D = \langle \Upsilon, \Upsilon_C, \Sigma, C \rangle$ is a tuple, where Υ is a (possibly many-sorted or order-sorted) domain signature that defines the concepts of the language, Υ_C is a constraints signature for auxiliary definitions, Σ is an (infinite) alphabet and C is a set of logic formulas. The term algebra $T_\Upsilon(\Sigma)$ over Υ generated by Σ defines the structures of the language, and the set of model realizations $R_\Upsilon = \mathcal{P}\left(T_\Upsilon(\Sigma)\right)$ is the power-set over this term algebra. The constraint signature Υ_C is an extension of Υ that contains all the necessary auxiliary symbols for expressing the well-formedness of the domain. Then, the well-formed models of a domain is the set of $\{r \in R_\Upsilon \mid \Psi(r), C \vdash \Psi(\mathit{wellformed})\}$, where Ψ is a function that converts model realizations into logic formulas, $\vdash$ represents deducibility, and *wellformed* $\in \Upsilon_C$ is a special constant tagging well-formed models [59].

An interpretation $\llbracket . \rrbracket \colon R_\Upsilon \to R_{\Upsilon'}$ is a mapping from the model realizations of a domain D to a domain D'. Such interpretations generalize model transformations,

and most importantly for us, they can provide behavioral semantics for language D, in which case $[\![.]\!]$ describes a semantic mapping.

3.3 Behavioral semantics

3.3.1 Denotational semantics

Christopher Strachey and Dana Scott [113, 114, 115, 122] developed denotational semantics to assign precise mathematical meaning to programs. Denotational semantics (also known as mathematical semantics) describes the semantics of a language by defining semantic equations (semantic functions) that map phrases and subphrases to mathematical objects. Compositionality is an important property of denotational semantics: it means that the semantics of a phrase is explained in terms of the semantics of its subphrases. This has many important consequences:

- Compositional specifications are easier to understand.
- Compositionality facilitates bottom-up reasoning based on structural induction.
- In a compositional specification, if two phrases denote the same mathematical object (i.e., they are semantically equivalent), they can be substituted for each other.

Another advantage of denotational semantics is its capability to describe a compilation process that can be leveraged for automated compiler generation [83, 101, 44].

Application to DSMLs In order to specify the denotational semantics of a DSML, we need the following constituents:

- a mathematical semantic domain that is sufficiently rich to capture behaviors of the models,

- a semantic function that maps the elements of the abstract syntax to the semantic domain.

There are numerous choices for the semantic domain, and choosing the ”right” domain is specific to each problem. Programming languages are usually mapped to λ-calculus. However, it would be extremely cumbersome to describe programming languages using pure λ-calculus. In order to gain conciseness and clarity in the specifications, we can choose higher-level semantic domains. For example, applied λ-calculus has pre-defined constants for numbers, Boolean values, operators and others. This raised level of abstraction greatly simplifies the specifications and improve their comprehensibility by removing uninteresting details. For the semantic specification of *concurrent processes* there are a number of possible choices: e.g., process calculi (e.g. CCS [80] and CSP [52]), Petri nets [96] and the Actor model [2]. For *embedded and CPS systems* we may map our languages to hybrid models, such as timed automata [4], hybrid automata [5, 49] or hybrid IO automata [73]. Finally, for *physical modeling languages*, we can use differential algebraic equations (DAE) to represent trajectories over continuous time and space, as we demonstrate in [119, 116].

Note that the graph structure of a modeling language can possibly cause problems with regard to the compositionality property. For modeling languages, we need to develop denotational specifications that do not contain loops: definitions are not allowed to directly or indirectly depend upon themselves.

Example As an example, we discuss the denotational semantics of a simple imperative language SIMP (a similar but simplified version of IMP languages [133, 108]) that contains natural numbers, Boolean values, variables, addition operator, if-then-else structure and while loop. The abstract syntax of the language is the following:

```
Statement ::= skip | Identifier:=Expression | Statement;Statement
    | if (BoolExpression) then Statement else Statement
    | while (BoolExpression) Statement.
Expression ::= Numeral | Identifier | Expression+Expression.
BoolExpression ::= Boolean | Expression==Expression
    | not BoolExpression.
```

The semantic domains (and metavariables) are:

$$
\begin{aligned}
b &\in \mathbb{B} = \{\mathit{tt}, \mathit{ff}\} \\
n &\in \mathbb{N} = \{0, 1, \ldots\} \\
+&: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\
=&: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B} \\
\sigma &\in \Sigma = [\texttt{Identifier} \rightarrow \mathbb{N} \cup \bot]
\end{aligned}
$$

where $\mathbb{B}$ is the set of Boolean values, $\mathbb{N}$ is the set of natural numbers, $+$ is a function that maps two natural numbers to their sum, $=$ is an equality predicate that maps pairs of natural numbers to Boolean values, and σ is a store function that maps identifiers to natural numbers or $\bot$, where $\bot$ denotes an undefined value (which makes σ a partial function). We use $[\texttt{Identifier} \rightarrow \mathbb{N} \cup \bot]$ to denote all the functions mapping identifiers to natural numbers or $\bot$.

Furthermore, assume that we have a store update function denoted $\sigma[x \mapsto s]$ that returns a new environment, where variable x equals s, and the rest of the store is intact:

$$
\sigma[x \mapsto s](y) = \begin{cases} \sigma(y) & \text{if } y \neq x \\ s & \text{otherwise} \end{cases}
$$

The semantic functions $\mathcal{S}$, $\mathcal{E}$ and $\mathcal{B}$ have the following signatures:

$$
\begin{aligned}
\mathcal{S}&: \texttt{Statement} \rightarrow (\Sigma \rightarrow \Sigma) \\
\mathcal{E}&: \texttt{Expression} \rightarrow (\Sigma \rightarrow \mathbb{N}) \\
\mathcal{B}&: \texttt{BoolExpression} \rightarrow (\Sigma \rightarrow \mathbb{B})
\end{aligned}
$$

Note how these functions define mappings from the abstract syntax to the semantic domains.

The semantic equations are the following:

$$\mathcal{S}[\![\texttt{x:=E}]\!]\sigma = \sigma[x \mapsto \mathcal{E}[\![\texttt{E}]\!]\sigma]$$

$$\mathcal{S}[\![\texttt{S1;S2}]\!] = \mathcal{S}[\![\texttt{S2}]\!] \circ \mathcal{S}[\![\texttt{S1}]\!]$$

$$\mathcal{S}[\![\textbf{if}\ \texttt{(EB)}\ \textbf{then}\ \texttt{S1}\ \textbf{else}\ \texttt{S2}]\!]\sigma = \\ \quad \textit{if}\ \mathcal{B}[\![\texttt{EB}]\!]\sigma\ \textit{then}\ \mathcal{S}[\![\texttt{S1}]\!]\sigma\ \textit{else}\ \mathcal{S}[\![\texttt{S2}]\!]\sigma$$

$$\mathcal{S}[\![\textbf{while}\ \texttt{(EB)}\ \texttt{S}]\!]\sigma = \textit{repeat}\ \mathcal{B}[\![\texttt{EB}]\!]\ \mathcal{S}[\![\texttt{S}]\!]\ \sigma \\ \quad \text{where}\ \textit{repeat} = \lambda t\ \lambda s\ \lambda\sigma(\textit{if}\ t\sigma\ \textit{then}\ \textit{repeat}\ (s\sigma)\ \textit{else}\ \sigma)$$

$$\mathcal{S}[\![\textbf{skip}]\!]\sigma = \sigma$$

$$\mathcal{E}[\![\texttt{Identifier}]\!]\sigma = \sigma(\texttt{Identifier})$$

$$\mathcal{E}[\![\texttt{Num}]\!]\sigma = n$$

$$\mathcal{E}[\![\texttt{E1+E2}]\!]\sigma = \mathcal{E}[\![\texttt{E1}]\!]\sigma + \mathcal{E}[\![\texttt{E2}]\!]\sigma$$

$$\mathcal{B}[\![\textbf{true}]\!]\sigma = \textit{tt}$$

$$\mathcal{B}[\![\textbf{false}]\!]\sigma = \textit{ff}$$

$$\mathcal{B}[\![\texttt{E1==E2}]\!]\sigma = \mathcal{E}[\![\texttt{E1}]\!]\sigma\ \textit{equals}\ \mathcal{E}[\![\texttt{E2}]\!]\sigma$$

$$\mathcal{B}[\![\textbf{not}\ \texttt{EB}]\!]\sigma = \textit{not}\ \mathcal{B}[\![\texttt{EB}]\!]\sigma$$

where the object language is distinguished from the meta-language with $[\![.]\!]$. Note that the meaning of the recursive *repeat* function is the least fixed-point solution of $YF = F(YF)$, where $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is the Y combinator in λ-calculus, and

$$F = \lambda f\ \lambda t\ \lambda s\ \lambda\sigma(\textit{if}\ t\sigma\ \textit{then}\ f(s\sigma)\ \textit{else}\ \sigma).$$

The existence and uniqueness of such a solution is provided by Scott's domain theory and the Kleene fixed-point theorem.

A good tutorial introduction with many examples to the theory of denotational semantics is presented by Tennent in [127] and by Slonneger in [120]. A more detailed explanation of semantic domains and denotational semantics is found in [39].

Note that the specification style presented for SIMP is called the direct semantics style. While this approach works for simple languages, many extensions have been proposed during the years:

- continuation semantics was proposed for handling jumps [123],
- monads and monad transformers were proposed for achieving modularity [81, 72],
- action semantics was introduced as a modular approach for denotational descriptions [82],
- VDM semantics was proposed for achieving simpler specifications by using a larger set of combinators [86].

3.3.2 Translational semantics

Translational semantics describes the behavior of a language by mapping its phrases to a target language that has well-defined semantics. If the target language is a programming language, this mapping defines a code generator.

The advantage of translational semantics is that the tools of the target language may be used for running simulations, performing formal analysis techniques and others. Its disadvantage is that the semantics of the source language is hidden in the semantics of the target language, and therefore, it is not explicitly formalized.

3.3.3 Operational semantics

Operational semantics describes a procedure that results in the semantics of the program when executed. There are different variants based on how the procedure is described. In this chapter, we discuss the Structural Operational Semantics, and in the next chapter, we refer to other methods (rewriting systems, Abstract State Machines).

Structural Operational Semantics

A well-known notation for operational semantics is the Structural Operational Semantics (SOS). SOS was proposed by Gordon Plotkin [97] in 1981 to define the operational semantics of programming languages. Interestingly, it was only 23 years later – in 2004 – that the original manuscript was finally published as a journal publication [98]. By this time, SOS has already become one of the most prominent specification styles.

As Plotkin notices in his "The Origins of Structural Operational Semantics" paper [99], the term 'structural' reflects that the SOS specifications are syntax-directed, therefore the behavior is specified in terms of the structure of the language, and the behavior function can be found by using structural induction over the specifications.

The original SOS specification is now known as small-step SOS. In contrast, what is now known as the big-step SOS, was developed by Gilles Kahn [61] and called natural semantics originally.

While the small-step semantics contains all the rules for the step-by-step execution of a program (model), the big-step semantics describes a much higher level abstraction: the overall result of executing it. Due to its abstract nature, big-step semantics looks similar to denotational semantics. Under the hood, there are differences though: while big-step semantics ignores the non-termination of programs, denotational semantics takes special care of such cases.

Small-step SOS SOS uses syntactic rules ("symbol pushing") to describe the transformation of programs and manipulation of data. Syntactic rules are based on the notion of Transition Systems (TSs). A TS is a structure $\langle \Sigma, \longrightarrow \rangle$, where Σ is a set of configurations and $\longrightarrow \subseteq \Sigma \times \Sigma$ is a transition relation. Transition relation $C_1 \longrightarrow C_2$ (infix form) expresses a transition from configuration C_1 to configuration C_2. The SOS syntactic rules are then expressed using inference rules of the form:

$$\frac{A_1 \ldots A_k}{A_0} \text{ [condition]},$$

where $A_0 \ldots A_k$ are transition relations. The meaning of this rule is that whenever the premises $A_1 \ldots A_k$ and [condition] hold, we can infer the consequence A_0.

For example, a configuration could be $\langle e, \sigma \rangle$, a pair of expression e and store σ, where $\sigma\colon \texttt{Ide} \to \texttt{Values}$ is a function that assigns a value to each identifier. Or, for languages with variable definitions, it is common to replace the store function with $\sigma\colon \texttt{Location} \to \texttt{Values}$, a function that assigns values to abstract locations, and defining an environment $\rho\colon \texttt{Ide} \to \texttt{Values} \cup \texttt{Location}$ that maps identifiers either to values (constants) or to locations (variables). Then, a configuration is a tuple $\langle e, \sigma, \rho \rangle$. For clarity, if the environment does not change in a rule, usually the alternative notation of $\rho \vdash \langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ is used instead of $\langle e, \sigma, \rho \rangle \longrightarrow \langle e', \sigma', \rho \rangle$.

Example For instance, the SOS semantics of the addition operator in our SIMP language is the following (here, σ is a store, E is an expression, n and m are natural numbers, x is an identifier):

$$\text{(ide)}\ \frac{}{\langle x, \sigma \rangle \longrightarrow \langle \sigma(x), \sigma \rangle}$$

$$\text{(left)}\ \frac{\langle E_1, \sigma \rangle \longrightarrow \langle E_1', \sigma \rangle}{\langle E_1 + E_2, \sigma \rangle \longrightarrow \langle E_1' + E_2, \sigma \rangle}$$

$$\text{(right)}\ \frac{\langle E, \sigma \rangle \longrightarrow \langle E', \sigma \rangle}{\langle n + E, \sigma \rangle \longrightarrow \langle n + E', \sigma \rangle}$$

$$\text{(add)}\ \frac{}{\langle n + n', \sigma \rangle \longrightarrow \langle m, \sigma \rangle}\ m \text{ is the sum of } n \in \mathbb{N} \text{ and } n' \in \mathbb{N}$$

The (ide) rule expresses how the value of an identifier is retrieved from the store; the (left) rule describes the evaluation order for the '+' operator (left-to-right); the (right) rule expresses that if the first argument is a number, the second argument may be rewritten; and finally the (add) rule defines how to evaluate '+' if both of its arguments are numbers.

As another example, here are the semantic specifications for the if-then-else structure:

$$\frac{\langle B, \sigma \rangle \longrightarrow \langle B', \sigma \rangle}{\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, \sigma \rangle \longrightarrow \langle \textbf{if } B' \textbf{ then } S_1 \textbf{ else } S_2, \sigma \rangle}$$

$$\frac{}{\langle \textbf{if true then } S_1 \textbf{ else } S_2, \sigma \rangle \longrightarrow \langle S_1, \sigma \rangle}$$

$$\frac{}{\langle \textbf{if false then } S_1 \textbf{ else } S_2, \sigma \rangle \longrightarrow \langle S_2, \sigma \rangle}$$

And the semantics of the while loop is:

$$\frac{}{\langle \textbf{while } B \textbf{ do } S, \sigma \rangle \longrightarrow \langle \textbf{if } B \textbf{ then } (S; \textbf{while } B \textbf{ do } S) \textbf{ else skip}, \sigma \rangle}$$

SOS suffers from modularity problems similar to the original denotational semantic specifications. Just as how monads and monad transformers are used for achieving modularity in denotational semantics, Modular Structural Operational Semantics (MSOS) [84] was introduced to achieve modularity in SOS specifications. The main difference between SOS and MSOS is that MSOS is based on labeled transition systems, where the labels describe the changes in the environment (store, etc.).

Natural semantics (big-step SOS) Kahn introduced natural semantics [61] as an alternative way for defining behavioral semantics. Kahn proposed the usage of axioms and inference rules (similar to natural deduction, hence the name: natural semantics) to define the semantics of the language phrases. Based on these rules, the semantics of a program can be inferred in an operational style.

Similar to the small-step SOS, a rule has premises and consequences, which are sequents written in the form of $C \Downarrow R$, denoting that evaluating configuration C results in configuration R.

In natural semantics, a rule is of the form:

$$\frac{C_1 \Downarrow R_1 \ldots C_k \Downarrow R_k}{C \Downarrow R} \text{ [conditions]},$$

where R and $R_i \ldots R_k$ are irreducible configurations. This is the main difference between SOS and natural semantics: while SOS rules produced intermittent configurations, in natural semantics the rules are describing final configurations.

Example To demonstrate the difference with the other semantic specifications, we define the natural semantics of the SIMP language. The natural semantics of addition is:

$$(\text{number})\ \frac{}{\langle n, \sigma \rangle \Downarrow \langle n \rangle}$$

$$(\text{identifier})\ \frac{}{\langle v, \sigma \rangle \Downarrow \langle \sigma(v) \rangle}$$

$$(\text{add})\ \frac{\langle E_1, \sigma \rangle \Downarrow \langle n_1 \rangle \qquad \langle E_2, \sigma \rangle \Downarrow \langle n_2 \rangle}{\langle E_1 + E_2, \sigma \rangle \Downarrow \langle m \rangle}\ m \text{ is the sum of } n_1 \in \mathbb{N} \text{ and } n_2 \in \mathbb{N}$$

The semantics of if-then-else is the following (note that the evaluation of Boolean expressions do not result in the change of state – they are side-effect free):

$$\frac{\langle B, \sigma \rangle \Downarrow \langle \textit{true} \rangle \qquad \langle S_1, \sigma \rangle \Downarrow \langle \sigma' \rangle}{\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, \sigma \rangle \Downarrow \sigma'}$$

$$\frac{\langle B, \sigma \rangle \Downarrow \langle \textit{false} \rangle \qquad \langle S_2, \sigma \rangle \Downarrow \langle \sigma' \rangle}{\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, \sigma \rangle \Downarrow \sigma'}$$

And the semantics of the while loop is:

$$\frac{\langle B, \sigma \rangle \Downarrow \langle \textit{true} \rangle \qquad \langle S, \sigma \rangle \Downarrow \langle \sigma' \rangle \qquad \langle \textbf{while } B \textbf{ do } S, \sigma' \rangle \Downarrow \sigma''}{\langle \textbf{while } B \textbf{ do } S, \sigma \rangle \Downarrow \sigma''}$$

$$\frac{\langle B, \sigma \rangle \Downarrow \langle \textit{false} \rangle}{\langle \textbf{while } B \textbf{ do } S, \sigma \rangle \Downarrow \sigma}$$

Natural semantics combines the advantages of denotational semantics and SOS: it provides good abstraction while retaining notational easiness. In most situations, natural semantics is a good choice for semantic specifications, which is further supported by the easiness of writing interpreters for the specifications in any functional or logic languages. However, there are some cases when natural semantics is inconvenient or impossible to use: for example, non-deterministic behavior or concurrent processes cause troubles. Furthermore, similar to small-step SOS, lack of modularity is another problem, which can be remedied using MSOS [84].

3.4 Conclusion

In this chapter, we surveyed some approaches for the behavioral specification of programming languages. In particular, we covered three types of semantics, which have different advantages and disadvantages:

- *Denotational semantics* is a concise, high-level representation for semantics. Its conciseness is also a disadvantage, since generally denotational specifications are very hard to understand. Its compositionality facilitates bottom-up reasoning and formal proofs through structural induction. While it is not necessary, denotational semantic specifications often determine an implementation, which can be leveraged to generate compilers based on the specifications.
- *Operational semantics* defines an interpretation of the language, therefore it can be easily used for writing interpreters, running simulations and performing model checking. Compared to denotational semantics, operational semantics is easier to understand; however, it is often more verbose. The disadvantage of operational semantics is that it is not capable of describing the semantics of modeling languages with behavior in the continuous-time domain.
- *Translational semantics* is executable and may be used for performing automated code generation to the target language. A distinct advantage of translational semantics is that the tools of the target language may be used for running simulations, performing formal analysis techniques and others. A hindrance is that the semantics of the source language is hidden in the semantics of the target language, and in most cases, it requires significant efforts to reverse engineer this semantics.

We will use these specification styles for the formalization of Cyber-Physical System (CPS) modeling languages. In particular,

- We will use the operational specification style to describe the behavior of computational languages in CPSs, e.g., the semantics of synchronous data-flow or statechart languages.

- We will extend the denotational specification style to describe CPS modeling languages. For this, we need to identify the mathematical objects, where the CPS language elements are mapped to, and provide *semantic units* (reusable semantic domains) for representing collections of these mathematical objects.
  - We define semantic units for physical modeling languages in the field of differential calculus (differential equations to represent continuous time and space trajectories), and for synchronous data-flow languages in the field of discrete calculus (difference equations).
  - We define semantic units for CPS integration languages that can represent the composition of continuous-time and discrete-time behavior: such as hybrid automata, or the composition of differential and difference equations using sample-hold operators.

Note that here we diverge from the traditional meaning of denotational semantics, since we map our languages to another languages, i.e., we describe translational semantic mappings. However, since there is a tight coupling between the abstract elements of the target domain and the corresponding mathematical objects, we consider them (pseudo-)denotational semantic specifications.

CHAPTER 4

FORMAL SPECIFICATION LANGUAGES FOR CPS DSMLS

In this chapter, we discuss several formal specification languages that have been used for specifying the semantics of modeling languages – their advantages and disadvantages, as well as their tool support.

First, we shall clarify the advantages of *formal* semantic specifications:

- **Unambiguity:** they provide an unambiguous and precise documentation for the language.
- **Reasoning:** a formal specification facilitates sound reasoning using the tools of mathematical logic.
- **Guidelines:** they provide guidelines for an implementation without specifying any particular implementation techniques.
- **Tool support:** formal specifications facilitate automated generation of language-based tools such as compilers, editors, inspectors, debuggers and visualizers [48].

Many formal languages have been promoted for formal specifications. They differ in data representation (relational structures [53], term algebra [55], etc.), their underlying logic (relational logic [53], rewriting logic [75], fixed point logic [55], higher-order logic [94], etc.) and the tool-support and formal analysis techniques they facilitate. Some of these languages are highly expressive but not executable, while others sacrifice expressiveness for executability.

In the modeling language context, a different classification is obtained by grouping approaches based on the location of the specifications [20]. In the weaving approach, meta-models and semantic specifications are woven together, which has the

benefit of encapsulated syntactical and semantical specifications. In the rewriting approach, the semantics is described by performing (graph) rewrites on the meta-model. The benefit is that the semantics is directly expressed in terms of the meta-model. In the translational approach, the meta-model is translated to an external specification language, and the semantics is defined in that language. The advantage is that we can use the available tools of the external language for performing formal analysis.

In the following, we discuss these approaches in detail, and provide an overview of some of the most well-known formal languages found in the literature. Note that there are other specification languages for semantic specifications (e.g., the K framework [109]), but we are not aware of their usage for the specification of Domain-Specific Modeling Languages (DSMLs), hence we do not discuss them in this chapter.

4.1 Specification approaches

4.1.1 Weaving approach

In the weaving approach, the behavior is woven into the abstract syntax by a meta-language that describes the operations of the model elements in the meta-model. The advantage of this approach is that both the syntax and the semantics are defined in the meta-model. Furthermore, since the operations are described in a meta-language, by choosing an executable meta-language the models are also executable.

KerMeta [91] is a representative of this approach, in which a statically typed meta-language is introduced for defining the operational semantics of meta-models. This action language is woven into a meta-data language using aspect-oriented modeling techniques, which results in an executable meta-language. The authors describe a case study using KerMeta for defining a Finite-State Machine (FSM) model.

Gargantini et al. [36] describe a specification based on Abstract State Machine (ASM). The authors weave ASM specifications into meta-models and compare this

technique to three ASM-based translational approaches. As a case study, they provide the specification of a FSM model.

Di Ruscio et al. [27] introduce a weaving approach using the XASM language for defining the dynamic semantics of Domain-Specific Languages (DSLs) using the Model-Driven Engineering (MDE) framework called Atlas Model Management Architecture (AMMA). In this work, the authors integrate the ASM language in AMMA, and use it as the transformation language between meta-models.

4.1.2 Rewriting approach

Alternatively, the behavioral semantics can be specified using rewriting systems. A rewriting system consists of a set of rewrite rules, each of which defines a mapping from the left-hand side of the rule to its right-hand side. Whenever the left-hand side of a rule matches a sub-phrase, it is substituted with the right-hand side of the rule. Often, rewrite systems are executed by applying these rewrite rules as long as there are applicable rules left. In other systems, control flow structures are used for defining the application order of the rules.

Wachsmuth [130] leverages Query/View/Transformation (QVT) relations [92] (an Object Management Group (OMG) standard for model transformations) to describe the structural operational semantics of DSMLs. In particular, they develop the operational semantics for Petri nets, as well as for a stream-oriented language for earthquake early warning systems.

In graph rewriting systems (graph transformation systems), the rules are specified using graph grammars and the pattern matching for the LHS is subgraph matching.

Agrawal, et al. [3] describe a semantic translation of Simulink Stateflow models to hybrid automaton models by defining a series of graph transformations using the graph transformation system GReAT [64, 10].

4.1.3 Translational approach

In the translational approach, the meta-model of a language is translated to a formal specification language and its semantics is described using the constructs of the specification language. An advantage is that the tools developed for the specification language can be used for performing formal analysis. A disadvantage of this approach is that the semantics is indirectly defined in an external language; therefore the user has to learn another language. Most of the work presented in the following section is based on the translational approach.

4.2 Specification languages

In this section, we discuss several specification languages for the specification of DSMLs. Note that the list is far from being exhaustive, but contains only the most prominent languages.

4.2.1 Constraint logic programming languages

FORMULA

FORMULA [56, 55] is a constraint logic programming language based on fixed-point logic over algebraic data types. Based on an initial set of facts specified using algebraic data types and a set of inference rules, FORMULA can deduce a set of final facts that is the least fixed-point solution for the specifications. Furthermore, given a partial model - a model with underspecified facts - and some constraints, FORMULA can find a completion of the model such that the constraints are satisfied, or return ‘unsatisfiable‘ if no such model exists. For this, FORMULA leverages Microsoft Z3 [26], a state of the art Satisfiability Modulo Theories (SMT) solver.

In FORMULA, domains are composed of data types and rules. A special constant called ‘conforms‘ is defined for representing well-formed models: if ‘conforms‘ is deduced by the inference engine, the model is well-formed. Furthermore, mapping

between domains is supported by FORMULA transformations that are sets of rules describing the mapping process. FORMULA comes with a bounded model checking tool that can be used for model checking sequences of transformations.

Jackson et al. [57, 60] use FORMULA as a formal language for specifying the structural semantics of DSMLs. Simko et al. [117, 116] use FORMULA for specifying the denotational semantics of a physical modeling language, a bond graph language. Furthermore, Simko et al. [119] describe both the structural and denotational specifications of a Cyber-Physical System (CPS) modeling language.

The advantage of FORMULA is that the language is designed for specifying the semantics of modeling languages; therefore, it provides comprehensible syntax for describing domains along with their structural semantics, as well as for describing behavioral semantic mappings with transformations.

Alloy

Alloy is a specification language based on first-order relational logic over first-order (flat) relational structures. An Alloy specification contains a set of signatures describing atoms, a set of declarations that defines relations over atoms, a set of facts (constraints that always hold), predicates and functions to define operations over the relational structure, and a set of assertions. The Alloy Analyzer tool can analyze these specifications by reducing them to Satisfiability (SAT)-formulas, which can be solved by the Kodkod [128] constraint solver. A significant limitation of this approach is that numbers are also mapped as atoms, and therefore, they are only evaluated in a limited scope.

Alloy has been used by many authors as a formal underpinning for graph transformations. In these works, the operational semantics is specified as graph transformations, but executed and analyzed in the Alloy framework.

Baresi and Spoletini [12] discuss how a graph transformation system specified in the AGG environment [126] can be encoded as an equivalent Alloy model, and demonstrate how to exploit Alloy's tools to answer bounded reachability questions with regards to the transformations.

Similarly, in the case study by Demirezen et al. [25], the operational semantics of a DSML is specified using a sequence of graph transformation rules modeled in AGG [126]. The authors describe how to manually encode the graph transformation rules in Alloy.

Although we are not aware of any work of expressing the structural semantics of DSMLs in Alloy, its relational algebra and the constraints could be leveraged to develop similar specifications as in FORMULA. A similar line of work (i.e., defining structural rules) is presented by Anastasakis et al. in [6]. UML2Alloy is a tool that can transform a subset of the static Unified Modeling Language (UML) class diagrams and Object Constraint Language (OCL) constraints to Alloy. The generated Alloy model can be used to discover design flaws in UML designs with the Alloy Analyzer tool.

Alloy is a powerful modeling language for software-based systems; however, it also has some limitations for describing the semantics of DSMLs. First, its flat relational structure is inconvenient for representing hierarchical structures, e.g., algebraic expressions. Second, in some scenarios – such as the specification of resource allocations – the lack of support for real numbers may be too limiting.

4.2.2 Rewriting logic

Rewriting logic [78] is a framework for expressing both the static and dynamic semantics of programming languages and concurrent systems. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ consists of an equational theory (Σ, E) and rewrite rules R. The equational theory describes the statics and the rewrite rules describe the dynamics of the system.

An equational theory (Σ, E) consists of function symbols Σ and equations E, which together define the semantics by their initial algebra. Non-deterministic systems are supported through the rewrite rules R that are of the form $r\colon t \to t'$, where r is a label, and $t, t' \in T_\Sigma(X)$ are elements of the term algebra over Σ and a finite set of variables X.

Maude

An implementation of rewriting logic is the Maude language [23, 24]. Maude's equational theory is a Membership Equational Logic (MEL), which is a generalization of order-sorted logic (which is a generalization of many-sorted logic, which is a generalization of unsorted logic). MEL is a sorted logic that supports partial ordering of sorts by subsort relations, subsort polymorphic overloading of operators, (conditional) membership axioms of the form $t : S$, asserting that t has sort S, and (conditional) equational axioms of the form $t = t'$ expressing the equality of terms t and t'. Maude extends MEL with a rewriting logic part that adds the notion of non-deterministic rewriting rules. Maude ships with a Linear Temporal Logic (LTL) model checker, and a real-time simulation tool that can be used for executing the specifications.

Boronat and Meseguer [19] use Maude for developing the algebraic semantics of the MetaObject Facility (MOF). Besides providing an unambiguous specification for MOF, such a formal semantics is executable, and can be used for formal analysis, such as consistency checks.

Romero et al. [107] use Maude to specify the static semantics of models and meta-models. In particular, they demonstrate how model subtyping, type inference, and metric evaluation can be specified and implemented in Maude.

Rivera and Vallecillo [103] describe the addition of operational semantics to models of DSLs in Maude. The specifications are used for running simulations, performing reachability analysis and model checking. In [104], Rivera et al. describe an upgraded version that is based on the core language of Maude. As part of the specifications, a sort is defined for each meta-meta-model element, such as for classes, attributes, references, and others. Then, the concepts of the meta-model are subsorts of these sorts, and a model is a set of Maude objects, where an object has a name, a sort, which defines its meta-model type, and a set of attributes and relations. Finally, rewriting rules are used for specifying the operational semantics of the DSML.

Rivera et al. [106] propose Maude to serve as a formal verification tool for graph rewriting transformations. They present a Maude code generator that is integrated

into a meta-modeling and model-transformation system called $AToM^3$. The code generator can automatically generate Maude specifications for both models and meta-models, and generate the rewriting rules corresponding to the graph transformations.

A specification framework called e–Motions is presented by Rivera et al. in [105] for describing the semantics of real-time Domain-Specific Visual Languages (DSVLs). In e–Motions, translational semantics is specified with the ATL Transformation Language that maps models to Real-Time Maude specifications. This mapping together with the behavioral semantics of Real-Time Maude defines the behavioral semantics of the language.

Egea and Rusu [29] present a Maude-based framework for specifying and checking model conformance based on OCL constraints. Rusu extends this work in [111] with the addition of operational semantics, which provides executable behavioral semantics for the models.

Maude is a well-known specification language that is a good fit for specifying the semantics of languages. The main disadvantage of Maude is that its syntax can easily become overwhelming. Nonetheless, as shown in the related works, Maude can be efficiently used for describing the structural and behavioral semantics of DSMLs.

4.2.3 Abstract State Machine languages

Abstract State Machine (also called dynamic or evolving algebras) is a widely used formal method developed by Yuri Gurevich for describing the semantics of sequential and non-sequential algorithms [40, 16]. ASM is a specification method based on the notion of states and updates: a computation is modeled as a sequence of states and state updates starting from an initial state. States are generic first-order structures, i.e., non-empty sets with functions and relations. Some of these functions are marked static, which means they are never updated. A state update describes the change in the interpretation of some dynamic functions, while static functions and the rest of the dynamic functions are left intact.

There are several specification languages for expressing the Abstract State Machine semantics of languages. Examples are the Abstract State Machine Language (AsmL) by Microsoft Research [79, 41], or the open source XASM language [7].

The research in ASM-based semantic specifications is abundant, it has found hundreds of applications in the specification of algorithms and programming languages, such as the specification of the C language [42], C# [17], Prolog [18], or the simulation semantics of SystemC [90].

Lately, ASM was proposed as the language for the operational semantics specification of DSMLs. Broadly, these attempts can be classified as translational and weaving approaches.

A translational approach using AsmL is introduced by Chen et al. [22], where a semantic anchoring framework is presented that defines the operational semantics of DSMLs by specifying a model transformation from the meta-model of the DSML to the meta-model of a semantic unit with well-defined semantics in AsmL. The model transformation is based on the graph transformation language GReAT [64, 10]. The operational semantics of the DSML is indirectly defined by the operational semantics of the semantic unit, which in turn is specified using the AsmL language. This semantic anchoring framework supports the composition of heterogeneous semantic units as discussed in [21].

Gargantini et al. [36] discuss different ASM based techniques for the semantic specification of modeling languages. They identify three translational techniques: semantic mapping, semantic hooking (anchoring) and semantic meta-hooking, and a weaving approach by augmenting the meta-model of the language with ASM constructs. As a case study, they provide the specification of a FSM language using these techniques.

Abstract State Machine is a powerful method for the operational semantic specification of languages. However, ASM does not support the specification of structural or denotational semantics; therefore, it provides only partial solution for the semantic specification challenge.

4.2.4 High-level specification languages

Z notation

The Z notation [121] is a mathematical notation for specifying computing systems. In Z, mathematical data types (sets, relations, functions, sequences, etc.) are defined based on the Zermelo-Fraenkel (ZF) set theory and predicate logic – together with operations that act on them. Specifications consist of schemes, where a schema contains a definition for a set of states, state invariants, operations and state update functions. Operations – respecting the invariants – are defined through the state update functions. A state update function is defined with predicates describing its pre- and post-conditions.

Esfahani et al. [31] introduce an activity-oriented DSML for modeling the functional and quality of service requirements of software systems using the Z notation. Evans et al. [33, 32] propose the Z notation for defining a precise formal semantics for UML class diagrams.

The main disadvantages of the Z notation are its unusual mathematical syntax and its lack of executability. Note however, that restricted sets of the language are executable, and several theorem prover and testing tools (testers and animators) are available for the language. A comprehensive comparison of Z-based tools is presented in [129].

PVS

The Prototype Verification System (PVS) [93, 94] is an interactive theorem prover based on higher-order logic with built-in proof tactics and automated theorem proving capabilities. PVS ships with a vast library of theories, which are reusable in subsequent proofs. Higher-order logic and the reusable library of theories make PVS capable of concisely expressing complex theorems.

Paige et al. [95] use PVS for developing the structural semantic specifications of the BON (Business Object Notation) language, which is a modeling language

similar to UML. Using the specifications, the authors use the PVS theorem prover to verify model conformance, and by adding semantics to the routines (pre- and post-conditions) they perform multi-view consistency checking.

The PVS language, similar to Z, is a powerful specification language with the same drawback: the specifications are generally not executable.

4.3 Conclusion

In this chapter, we discussed several formal languages for semantic specifications:

- FORMULA is a specification language based on constraint logic programming over algebraic data types that was specifically designed for the specification of modeling languages. FORMULA specifications are executable, and it has bounded model checking and model finding tools. Furthermore, there is a straightforward representation of models and meta-models in FORMULA.
- Alloy is a specification language based on constraint logic programming and relational structures. Alloy's advantages are its executability and support for model finding. Its main disadvantage is the lack of support for infinite domains.
- Maude is a well-known high-level specification language based on rewriting logic and MEL. Maude is executable and it has tools for performing model checking and running real-time simulations. Maude is a capable language that can be used for specifying the structural, operational and denotational semantics of DSMLs.
- ASM is a formal method for writing operational specifications; however, ASM lacks features for structural and denotational semantic specifications. The AsmL language (an implementation of ASM) has an execution engine, an explicit-state model checker and a conformance checker that compares implementations to specifications through automated testing.
- PVS and Z are generic high-level specification languages with support for interactive theorem proving; however, they are not executable.

An important aspect of a formal specification language is whether the specifications written in the language are analyzable and executable. Arguably, it is hard to gain confidence in a specification without extensive analysis and testing. Daniel Jackson, author of the Alloy specification language, writes the following in his book [53] (p. XIII):

> The experience of exploring a software model with an automatic analyzer is at once thrilling and humiliating ... Then the sense of humiliation sets in, as you discover that there's almost nothing you can do right. What you write doesn't mean exactly what you think it means. And when it does, it doesn't have the consequences you expected ... I now cringe at the thought of all the models I wrote (and even published) that were never analyzed, as I know how error-ridden they must be.

For this reason, we prefer specification languages with support for the following functions:

- Checking whether a model is well-formed according to its structural semantics.
- Evaluating the denotational semantic mapping to compile a model to its semantic domain.
- Simulating a model based on its operational semantics. Performing model checking based on some specifications (e.g., LTL specifications).
- Automatically finding well-formed models that satisfy some structural and/or behavioral properties (model finding).

Besides executability, there are three important questions that can help us choosing a language for the semantic specification of CPS DSMLs:

- Does the language support the concise and comprehensible representation of DSMLs and their models?
- Does it support the specification of structural semantics, as well as different types of behavioral semantics (e.g., denotational and operational)?

- What is the tool support for the language? Does it support model conformance checking, model finding, testing and model checking?

	FORMULA	Alloy	Maude	ASM	PVS	Z
Structural Semantics	x	x	x		x	x
Denotational Semantics	x	x	x		x	x
Executable Operational Sem.	x	x	x	x		
Model conformance checking	x	x	x			
Bounded Model checking	x	x	x	x		
Model finding	x	x				

Table 1: Comparison of different specification languages for DSML specification.

Table 1 compares the specification languages discussed in this chapter based on these criteria. The marked cells indicate that either the language provides language/-tool support for the given feature, or developing the feature is straightforward using the built-in features of the language. Based on these criteria, FORMULA, Maude and Alloy are probably the best choices for the semantic specification of CPS DSMLs. In the rest of this thesis, we will define a language extending FORMULA with additional language constructs to help the development of concise behavioral specifications for CPS modeling languages.

CHAPTER 5

A FORMAL LANGUAGE FOR SEMANTIC SPECIFICATIONS

In this chapter, we introduce a specification language that can efficiently describe the behavioral semantics of modeling languages. The language is based on the Microsoft FORMULA language but extends it with useful features for behavioral semantic specifications.

5.1 Introduction to ForSpec

ForSpec is a logic-based language for writing formal specifications. The syntax and semantics of ForSpec largely overlap with FORMULA [56, 55], a constraint logic programming language developed at Microsoft Research based on fixed-point logic over algebraic data types. We discuss the language in two parts: first, we introduce the overlapping concrete syntax with FORMULA. The abstract syntax and semantics of these elements are the same as in FORMULA [58]. In the second part, we introduce the new syntactic elements and describe their semantics.

5.1.1 Core language

The `domain` keyword specifies a domain (analogous to a meta-model), which is composed of type definitions, data constructors and rules. A model of the domain consists of a set of *facts* (also called initial knowledge) that are defined using the data constructors of the domain. The well-formed models of the domain are defined with the `conforms` rules. Given a model, if the constraints of the `conforms` rules are satisfied by the least fixed-point model obtained from the initial set of facts by applying the domain rules, the model is said to conform to its domain.

ForSpec has a complex type system based on built-in types (`Natural`, `Integer`, `Real`, `String`, and `Bool`), enumerations, data constructors and union types. Enumerations are sets of constants defined by enumerating all their elements; for example, `bool ::= {true,false}` denotes the usual 2-valued Boolean type.

Union types are unions of types in the set-theoretical sense, i.e., the elements of a union type are the union of the elements of the constituent types. Union types are defined using the notation of `T ::= Natural + Integer`, which defines type `T` as the union of natural and integer numbers; that is, type `T` denotes the integer numbers.

Data constructors can be used for constructing algebraic data types. Such terms can represent sets, relations, partial and total functions, injections, surjections and bijections. Consider the following type definitions:

```
A ::= new (x:Integer, y:String).
B ::= fun (x:Integer -> y:String).
C ::= fun (x:Integer => y:A).
D ::= inj (x:Integer -> y:String).
E ::= surj (x:Integer -> y:String).
F ::= bij (x:A => y:B).
G ::= Integer + H.
H ::= new (Integer, any G).
I ::= (x:Integer, y:String).
```

Data type `A` defines `A`-terms by pairing `Integer`s and `String`s (and `A` also stands for the data constructor for this type; for example, `A(5,"f")` is an `A`-term), where the optional `x` and `y` are the accessors for the respective values. Data type `B` defines a partial function (functional relation) from `Integer`s to `String`s. Similarly, `C` defines a total function from `String`s to `A`-terms, `D` defines a partial injective function, `E` defines a partial surjective function, and `F` defines a bijective function between `A`-terms and `B`-terms. Type `G` is the union of the integers and `H`, where `H` is a data type composed of an integer and a `G` term. Note that `G` and `H` are mutually dependent, which would cause an error message during static type checking. In order to avoid the error message, we use the `any` keyword in the definition of `H`.

While the previous data types (and constructors) are used for defining initial facts in models, derived data types are used for representing facts derived from the initial

knowledge by means of rules. For example, derived data type `I` defines terms over pairs of `Integer`s and `String`s.

Set comprehensions are defined in the form of `{head|body}`, which denotes the set of elements formed by `head` that satisfies `body`. Set comprehensions are used by built-in operators such as `count` or `toList`. For instance, given a relation `Pair ::= new (State,State)`, the expression `X is State, n = count({Y|Pair(X,Y)})` counts the number of states paired with state `X`.

Rules are used for deducing constants and derived data types. Rules have a left-hand side part called head, and a right-hand side part called body. The semantics of a rule is that whenever the body can be made satisfied (its variables can be substituted such that the resulting term is a fact in the model), then the head (after substituting with the same variable values) is also added to the facts. Negation (`no` keyword) is supported in the body by the restriction that the rules must be stratified in this case.

For example, we can calculate all the paths between nodes in a graph with the following specifications:

```
// node is a type that consists of an integer
node ::= new (Integer).
// edges are formed from pairs of nodes
edge ::= new (node,node).
// path is a derived data type generated by the following rules:
path ::= (node,node).
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z), edge(Z,Y).
```

In order to help writing multiple rules with the same left-hand side term, the semi-colon operator is used: its meaning is logical disjunction. For instance, in `A(X) :- S(X); T(X)` any substitution for `X`, such that `S(X)` or `T(X)` is derivable, results in the deduction of `A(X)`.

Type constraint `x:A` is true if and only if variable `x` is of type `A`, while `x is A` is satisfied for all derivations of type `A`. Similarly, `A(x,"a")` is a type constraint, which is satisfied by all substitutions for variable x such that the resulting ground term is a member of the knowledge set (note that the second sub-term is already grounded in

this example). Besides type constraints, ForSpec supports relational constraints, such as equality of ground-terms, and arithmetic predicates (e.g., less-than, greater-equal, etc.) over reals and integers. The special symbol _ denotes an anonymous variable, which cannot be referenced anywhere else.

Domain composition is supported by the `extends` and `includes` keywords. Both denote the inheritance of all types, data constructors and rules, but while `domain A extends B` ensures that all the well-formed models of `A` are well-formed models of `B`, definition `domain A includes B` may contain well-formed models in `A`, which are ill-formed models of `B`.

Namespaces are used for handling multiple definitions with the same name in different ancestor domains. For example, `domain A extends b::B` uses the name `b` for referring elements of `B`. In `A`, we can refer to these elements by inserting a dotted qualification `b.` in front of the type identifiers defined in domain `B`.

Finally, transformations define rules for creating output models from a set of input models and parameters. Transformations are specified as sets of rules. The semantics of a transformation is simple: if a non-derived data type of the output domain is deducible using the transformation rules, it will be a fact in the output domain. The syntax of a transformation has the following form:

```
transform name (list of inputs) returns (list of outputs)
{
  local type declarations
  rules
}
```

Here, the list of inputs consist of namespaced domain references (e.g., `in1::Stateflow` refers to the Stateflow domain by the name `in1`) and elementary arguments (e.g., `k:Integer` is an integer variable named `k`, which can be accessed by `%k` in the transformation rules). Similarly, the outputs are namespaced domain references. We can refer to elements in different namespaces by using the dotted notation discussed above.

Transformation rules have the same syntax and semantics as in domains with the only exception that a special type _ is introduced for each output domain that defines

an identity constructor for its types. The identity constructor is useful for copying terms from the input domains to the output domains.

For instance, the following transformation describes the firing of a Finite-State Machine (FSM), where all the data constructors belonging to `in.frame` are automatically copied to the output domain:

```
transform fire (in::FSM, trigger:String) returns (out::FSM)
{
  // Copy the frame data (static structure of FSMs)
  out._(X) :- X is in.frame.
  // Take triggered transition
  out.currentState(Next) :- in.currentState(C), in.Transition(C,
   %trigger, Next).
  // If no transition triggered, take stuttering step
  out.currentState(C) :- in.currentState(C), no in.Transition(C,
   %trigger, _).
}
```

5.1.2 Full language

ForSpec extends the core language with goal-driven and functional terms, semantic functions and semantic equations.

Goal-driven types The grammar rule for goal-driven type declarations is the following:

```
<gd-def> ::= <id> "::=" "[" <field>+ "=>" <field>+ "]".
```

For example, `F ::= [lhs:Integer,rhs:Integer => Integer]` defines a goal-driven type F as a tuple of three integers (the first two of which are accessed by the names *lhs* and *rhs*). Furthermore, it automatically defines a trigger type `F ::= (Integer,Integer)` as a pair of integers (the arguments *lhs* and *rhs* before the arrow) that triggers the evaluation of the goal-driven type. Since both the goal-driven type and its trigger type have the same names, ForSpec uses the number of arguments to decide which type

a term belongs to. Because the trigger type has always less arguments than the corresponding goal-driven type, the number of arguments uniquely defines the type of a term.

Besides the regular use of data constructors (e.g., `F(2,3,5)`), goal-driven terms can be also written according to the following syntax:

```
<gd-term> ::= <id> "(" <term>+ ")" "=>" (<term> | "(" <term>+ ")").
```

For instance, `F(2,3) => 5` and `G(1) => (2,3)` are valid goal-driven terms. Note that the `=>` operator is only a syntactic sugar in these cases, and the examples are equivalent to `F(2,3,5)` and `G(1,2,3)`.

The specialty of these goal-driven terms is the following:

- Whenever they appear on the left-hand side of a rule, the corresponding trigger term is appended to the right-hand side; i.e., the inference of a goal-driven term is dependent on its trigger term.
- Whenever they appear on the right-hand side of a rule, the corresponding trigger term is extracted as the head of a new rule that has all the constraints of the left-hand side of the original rule up to the point of the goal-driven term under question. This means that whenever a rule is dependent on a goal-driven term, a rule is generated for deriving the corresponding trigger term.

In order to describe the formal semantics of a goal-driven term, we will inject it in the semantics of the core language. Provided the semantics of the core language is given in terms of a knowledge-set K [58], the semantics of goal-driven rules is the following (note that goal-driven terms can be used only in rules that are free of set

comprehensions and negations):

$$\mathcal{R}[\![t \leftarrow \mathbf{p}]\!]n \stackrel{def}{=} \forall \mathbf{x}. \left(\left(\mathcal{G}[\![t[\mathbf{x}]]\!] \wedge \bigwedge_{p_i \in \mathbf{p}} \mathcal{R}[\![p_i[\mathbf{x}]]\!]n \right) \Rightarrow K(t[\mathbf{x}], n) \right) \wedge$$
$$\bigwedge_{p_i \in \mathbf{p} \wedge gd(p_i)} \mathcal{R}[\![tr(p_i) \leftarrow \{p_1, \ldots, p_{i-1}\}]\!]n$$
$$\mathcal{R}[\![F(t)]\!]n \stackrel{def}{=} \exists i.(i \leq n \wedge K(t, i))$$
$$\mathcal{R}[\![r(t, t')]\!]n \stackrel{def}{=} r(t, t')$$
$$\mathcal{G}[\![t]\!] \stackrel{def}{=} \begin{cases} \mathcal{R}[\![F(tr(t))]\!]n & \text{if } gd(t) \\ \top & \text{otherwise} \end{cases}$$

Here, $\phi[\mathbf{x}]$ refers to the substitution of $\mathbf{x}$ for the variables of the corresponding rule in term/predicate ϕ, $gd(\phi)$ is true iff ϕ is a goal-driven term or find-predicate, $tr(\phi)$ refers to the trigger term corresponding to ϕ, $F(t)$ is a find-predicate for term t, and $r(t, t')$ is a binary relation between terms. In the first line, the semantic function $\mathcal{G}[\![]\!]$ is used for making goal-driven heads dependent on their corresponding trigger term, and the second line extracts the rules for deriving the trigger terms for the body of the rule.

Functional terms Besides providing a goal-driven evaluation scheme, goal-driven trigger types have another use in functional terms. A goal-driven trigger term can be used as a function application, in which case its semantics is the result of its evaluation. For instance, if we have a goal-driven type `add ::= [expr, expr => Integer]` that evaluates to the addition of expressions, then the term `add(X,add(Y,5)) => T` is equivalent to `add(Y,5) => Z, add(X,Z) => T`, where `Z` is a variable not used anywhere else. This automatic unfolding of the internal trigger type (`add(Y,5)`) is a useful feature for writing behavioral specifications.

The semantics of functional terms is the following (assuming t and s are goal-driven terms):

$$\mathcal{R}[\![F(t(\mathbf{a}, s(\mathbf{b})))]\!]n \stackrel{def}{=} \mathcal{R}[\![F(t(\mathbf{a}, \mathbf{nv}))]\!]n \wedge \mathcal{R}[\![F(s(\mathbf{b}, \mathbf{nv}))]\!]n$$

where **nv** is a set of new variables that is not used anywhere else in the specifications, and **a** and **b** are the other arguments of s and t.

Union type extension ForSpec supports the extension of existing union type declarations with additional components. This is especially important in language design, since it facilitates the composition of languages.

For example, consider the following domain for describing expressions:

```
domain Equations
{
  expr ::= var + Real + op.
  op   ::= neg + add.
  var ::= new (name:String).
  neg ::= new (any expr).
  add ::= new (any expr, any expr).
}
```

If we wanted to create a language that supports multiplication, we would need to copy these specifications and add the multiplication operator. This results in code duplication, which can be avoided by using union type extension.

In ForSpec, the extended domain can be defined the following way:

```
domain Equations_Ext extends Equations
{
  op += mul.
  mul ::= new (any expr, any expr).
}
```

Here, the first line expresses that we add `mul` to the `op` union type. The resulting `Equations_Ext` domain is equivalent to the following domain:

```
domain Equations_Ext
{
  expr ::= var + Real + op.
  op   ::= neg + add + mul.
  var ::= new (name:String).
  neg ::= new (any expr).
```

```
  add ::= new (any expr, any expr).
  mul ::= new (any expr, any expr).
}
```

Union type extension improves the quality of the specifications by avoiding code duplication and by helping the language designer to use existing language components.

Semantic functions In order to support writing denotational semantic specifications, ForSpec introduces syntactic elements for defining semantic functions. A semantic function declaration has the following form:

```
<sem-func> ::= <id> ":" <field>+ "->" <field>+
```

The semantics of such a semantic function has two components: first, it declares a data type of the same name; second, it creates rules for extracting information from the semantic functions as discussed below. For example, the semantic function `name : dom_types -> codom_types` declares a data type equivalent to `name ::= [dom_types => codom_types]`, and the generated rules extract every possible instantiations of the `codom_types` over which the function ranges for a concrete model.

Consider a semantic function $p : x_1, ..., x_k- > x_{k+1}, ..., x_m$. The semantics of the generated rules is the following:

$$\mathcal{R}[\![p\colon x_1,\ldots,x_k \to x_{k+1},\ldots,x_m]\!]n \stackrel{def}{=} \forall \mathbf{x}.\forall i \in [k+1,m].\,(\mathcal{R}[\![F(p(\mathbf{x}))]\!]n \Rightarrow K(x_i,n))$$

In words, for every derived term of type p, the rules derive each of the sub-terms found in the co-domain of p.

Semantic equations ForSpec contains syntactic elements for writing semantic equations. The form of a semantic equation is the following:

```
<sem-eq> ::= <id> "[[" <id> "]]" <term>* "=" <term>+ ["where" <rule-body>+]
```

For example, the following specification defines the semantic equation for the add operator:

```
add  ::= new (expr,expr).
expr ::= add + ...
S : expr -> Integer.
S [[add]] = summa
where summa = S [[add.lhs]] + S [[add.rhs]].
```

Note that add is a data type in the domain; still, it is used as a variable on the right-hand side of the where keyword. This is possible due to the semantics of the semantic equations as follows. Consider a semantic equation

```
S [[Elem]] Args = M_1 M_2 ... M_k where C
```

The semantic equation defines how semantic function S maps element Elem and arguments Args to elements M_1, M_2 ... M_k, whenever constraints C hold. The semantics of the equation is the following:

$$\begin{aligned}
&\mathcal{R}[\![S(Elem, Args) = M_1 M_2 \ldots M_k \text{ where } C]\!]n \stackrel{def}{=} \\
&\quad \forall \mathbf{x}.\forall y.\forall i \in [1, k]. \\
&\qquad (\mathcal{R}[\![y : Elem]\!]n \wedge \\
&\qquad \mathcal{R}[\![F(y)]\!]n \wedge \\
&\qquad \mathcal{R}[\![C[\mathbf{x}]]\!]n) \Rightarrow K(S(y, Args, M_i), n)
\end{aligned}$$

In words, for every ground-term y of type $Elem$ (note that $Elem$ is an element of the meta-model, thus a type of the model) that is found in the knowledge-set we generate a mapping from y and $Args$ to each of the M_i-s.

Pretty-printing Note that the documentation generator will pretty-print the semantic functions and equations, so the previous example will have the following form in the documentation:

```
add  ::= new (expr,expr).
expr ::= add + ...
𝒮 : expr → Integer.
```

```
𝒮⟦add⟧ = summa
where summa = 𝒮⟦add.lhs⟧ + 𝒮⟦add.rhs⟧.
```

Example In order to demonstrate the use of the previous syntactic elements, consider the following example:

```
domain ExprLanguage
{
  expr ::= add + sub + Integer.
  add ::= (lhs:expr, rhs:expr).
  sub ::= (lhs:expr, rhs:expr).
}
domain Integers
{
  int ::= (Integer).
}
transform ExprSemantics (in::ExprLanguage)
returns (out::Integers)
{
  // Semantic function mapping expressions to integers.
  expr_sem : expr -> int.
  // Semantic equation for integers.
  expr_sem[[Integer]] = int(Integer).
  // Semantic equation for addition operator using functional rules:
  expr_sem[[add]] = summa
  where
    expr_sem[[add.lhs]] = int(n),
    expr_sem[[add.rhs]] = int(n'),
    summa = int(n+n').
}
```

Oftentimes, denotational mapping works by mapping a single concept of the language to several concepts of the semantic domain. In order to support the concise representation of such multi-mappings, ForSpec supports the specification of multiple terms on the right-hand side of a denotational equation separated by white-spaces.

For example,

```
S [[ConnectedPower]] =
  eq(sum("CyPhyML_powerflow",flow1.id), 0)
  addend(sum("CyPhyML_powerflow",flow1.id), flow1)
  addend(sum("CyPhyML_powerflow",flow1.id), flow2)
  eq(effort1, effort2)
where
  x = ConnectedPower.src, y = ConnectedPower.dst, x != y,
  DesignElementToPortContainment(cx,x), cx:Component,
  DesignElementToPortContainment(cy,y), cy:Component,
  PP [[x]] = (effort1,flow1),
  PP [[y]] = (effort2,flow2).
```

defines four semantic equations (lines 2–5), each of which defines the semantics of the `ConnectedPower` element with the same constraints on the right-hand side of the `where` keyword.

5.2 Semantic specification in ForSpec/Formula

In this section, we explore four approaches for writing behavioral specifications in ForSpec. Furthermore, for each approach, we identify the possible applications of the symbolic analysis techniques provided by the FORMULA engine.

5.2.1 Operational semantics in ForSpec

Transformation-based operational semantics The operational semantics of a language can be easily formalized in ForSpec (and in FORMULA as well) using transformations. Given the abstract syntax of a language (a domain in ForSpec), first, we enrich it with the details of the execution environment. Then, the operational semantics is an endogenous transformation for this domain.

For example, given the following domain for FSMs:

```
domain FSM
{
  State ::= new (String).
```

```
  Event ::= new (String).
  Transition ::= fun (State, Event -> State).
  InitialState ::= new (State).
}
```

We can extend this domain to an executable FSM domain by adding a data constructor for representing the current state:

```
domain Executable_FSM extends FSM
{
  CurrentState ::= new (State).
  // ensure that there is exactly one current state
  conforms count({Y | CurrentState(Y)}) = 1.
}
```

The transformation describing the operational semantics is the following:

```
transform fire (in::Executable_FSM, event:String) returns
    (out::Executable_FSM)
{
  // copy the frame data
  out._(X) :- X is in.FSM.Data.

  // successful firing
  out.CurrentState(Next) :-
    in.CurrentState(Current),
    in.Transition(Current, Event(%event), Next).
}
```

Such a specification can be used with the symbolic execution engine of FORMULA to answer the following question: given an input model, are there any events that lead to an ill-formed output model? In the FSM example, FORMULA could find such an event, if the input model has any states, from which an event does not trigger any transitions. In this case, the output model would not have a current state, and therefore it would be an ill-formed `Executable_FSM` model.

Rule-based operational semantics Describing rule-based operational semantics in ForSpec is more complicated. In this case, a transformation consists of many

smaller steps – each one expressed as a rule. While in the transformation-based operational semantics we had only two environments – the a priori and a posteriori environments for the transformation – in the rule-based operational semantics each rule produces a new environment.

As an example, consider a simple expression language with side effects:

```
domain SIMP
{
  ...
  // An algebraic data type for representing sequents.
  expr ::= [Expression, Environment => Natural, Environment].
  // Axiom schema for (number).
  expr(n,env) => (n,env) :- n:Natural.
  // Axiom schema for (identifier). Retrieve variable from environment.
  expr(v,env) => (n,env) :- v:Variable, retrieve(env,v,n).
  // Rule for (add).
  expr(Add(E1,E2),env) => (m,env'') :-
    expr(E1,env) => (n,env'), n:Natural,
    expr(E2,env') => (n',env''), n':Natural,
    m = n + n'.
}
```

The main difference compared to the transformation-based operational semantics is that in this case each rule is evaluated with respect to its own environment, and therefore the whole environment needs to be represented as a single algebraic data type. Furthermore, to support working with such environments, we also need auxiliary goal-driven functions that operate on them: such as the *retrieve* function in the example.

For both the transformation-based and the rule-based cases, there are many ways to leverage the specifications:

- **Symbolic model checking:** by attaching a monitoring language to the original language and specifying its semantics in the same operational style, the model finder can perform bounded model checking on a model by finding an input vector that produces a counterexample for a safety specification.

- **Automated test-case generation:** given a coverage requirement (e.g., modified condition/decision coverage [47]), the model finder can generate test vectors that exercise the required parts of the specification [110].

- **Non-determinism of the specification:** the model finder can be used to generate configurations that are reducible to different conclusions. If no such configuration exists, the specification is deterministic.

- **Existence of stuck configurations:** a stuck configuration is a non-final configuration, which has no applicable rules. If such configurations exist, we can use the model finder to find them.

5.2.2 Translational and denotational semantics

One of the most straightforward semantic specification styles in ForSpec is the translational semantic specification. In this case, ForSpec is used as a rewriting system between different abstract syntaxes (domains). Hence, for developing translational specifications, we need domains describing the meta-model of the source and target languages, and a transformation that translates models of the source language to the target language.

Based on the "distance" of the semantics of the source and target languages, the translation could be simple element-wise mapping, or more complicated rewriting rules. For example, a simple translational semantics is described by the following transformation:

```
transform TranslateESMoL (esmol::ESMoL_statechart)
returns (sf::Matlab_Stateflow)
{
  map_ts : esmol.TransStart -> sf.transition + sf.transition_type
    + sf.transition_order.

  // The translational semantics of ESMoL TransStart element.
  map_ts [[TransStart]] =
    sf._(m_trans)
    sf.transition_type(m_trans, DEFAULT)
```

```
    sf.transition_order(m_trans, T.Order)
  where
    // Find the transition with TransStart as its source
    T is esmol.Transition, T.Src = TransStart,
    // Find the parent state of the TransStart element
    esmol.StateToTransConnectorContainment(parent,TransStart),
    // Construct the corresponding transition
    m_trans = sf.transition(
                       map_state [[parent]],
                       map_node [[T.Src]],
                       map_node [[T.Dst]],
                       map_event [[T.Event]]).
  ...
}
```

This excerpt demonstrates how the `TransStart` concept of the Embedded Systems Modeling Language (ESMoL) [100] is translated to a default `transition` element of Matlab Stateflow. Function `map` represents the mapping from elements of the source domain to the target domain. A `TransStart` element is mapped to a `sf.transition` element along with `sf.transition_type` and `sf.transition_order` attributes.

There are important questions that can be answered by performing model finding on translational semantics:

- **Completeness:** are there any elements in a well-formed source model that are not mapped to any elements in the target domain? It is tricky to answer this question, since beside symbolically executing the behavioral specifications, we also need to consider the structural semantics of the language. For example, ESMoL contains a definition for history nodes, but they are not supported by the tools of ESMoL at the moment, and they are forbidden in well-formed models. Therefore, the specifications are complete, even though we do not specify the mapping of these history nodes.

- **Uniqueness:** are all the mappings unique? In many cases it indicates an error if multiple rules map a single concept. We can use model finding to explore these errors.

Note that the same approach can be used to specify the denotational semantics of a language, as we demonstrate in [116, 119]. In order to describe the denotational semantics, first, we need to define a semantic domain that is capable of describing the semantics of the language; second, we need to define semantic equations that relate the abstract syntax of the language to the semantic domain. As an example, consider the denotational semantics of a bond graph language (see Chapter 7 for details):

```
domain DAE
{
  term ::= cvar + Real + mul + ...
  cvar ::= new (name:String,id:Integer).
  mul  ::= new (any term,any term).
  eq     ::= new (term,term).
  diffEq ::= new (term,term).
  ...
}
transform BondgraphSemantics (graph::BondGraph)
returns (dae::DAE)
{
  // Semantic function for (B)onds
  B : graph.Bond -> dae.cvar, dae.cvar.
  // Semantic function for (N)odes
  N : graph.Node -> dae.eq.

  // The semantics of a bond is a variable pair (effort and flow).
  B [[Bond]] =
    (dae.cvar("effort",Bond.id),
     dae.cvar("flow",Bond.id)).

  // The semantics of a resistance R is an equation over the variables of its bond.
  N [[R]] = dae.eq(EffortVar, mul(FlowVar,R.Value))
  where
    // Find the bond of the resistance R
    graph.connects(Bond,R),
    B [[Bond]] = (EffortVar,FlowVar).
  ...
}
```

In practice, developing denotational semantics is a highly error-prone task, where tool support can greatly improve the consistency of a specification. On one hand, static type checking of the semantic functions filters out many problems. On the other hand, we can leverage model finding to provide further help for the designer:

- **Completeness:** if there are any undefined (not mapped) syntactic elements, the model finder can find them. This is a non-trivial problem, because the mappings are usually defined with respect to some constraints, and we need to answer if the union of all the mappings cover all the cases.

- **Consistency:** if the model finder can find a syntactic element that is defined twice with different semantics, the specification is inconsistent. Note that this is not as easy as checking the left-hand side of the semantic equations, because a syntactical element may be defined in several rules that only differ in their constraints on their right-hand side.

5.3 Conclusion

In this chapter, we introduced ForSpec, a specification language for the structural and behavioral specifications of Domain-Specific Modeling Languages (DSMLs). We discussed in details how the operational, translational and denotational semantics of a language can be formalized using ForSpec.

In order to avoid confusion about the possible uses of these specifications, Table 2 summarizes the different use cases for different modeling languages and specification styles.

Language type	Specification type	Usage
Any	Structural	Model conformance checking Model finding based on some specs
Any	Translational	Compiler to a different language
Computational	Operational	Interpreter Generating Traces Bounded model checking Model coverage Checking for stuck configurations Checking for non-deterministic specs
	Denotational	Compiler (e.g., to λ-calculus) Code generator to functional languages Checking for completeness of specs Checking for consistency of specs
Physical	Denotational	Compiler (e.g., to differential equations) Code generator to equation-based languages Checking for completeness of specs Checking for consistency of specs

Table 2: Use cases for the ForSpec specifications.

CHAPTER 6

REUSABLE SEMANTIC UNITS FOR FORMALIZING THE DENOTATIONAL SEMANTICS OF CPS MODELING LANGUAGES

In this chapter, we define a set of reusable semantic units for the denotational semantic specification of CPS modeling languages. We will use these semantic units in the following chapter to specify the behavioral semantics for several CPS modeling languages.

6.1 Semantic anchoring

Specifying the behavioral semantics using semantic mappings is a challenging and time-consuming task. In order to help this process, Chen et al. [22] introduced semantic anchoring earlier. In the semantic anchoring framework, the language designer anchors the semantics of the language to a previously defined semantic unit (a language with well-defined semantics). By maintaining a library of semantic units and reusing them for the specification of several languages, the cost of writing a specification can be greatly reduced.

A semantic unit $\mathcal{L}_{SU}$ is a language with a well-defined semantic mapping M_{Si} as shown in Fig. 1. By specifying the semantic mapping (anchoring) M_S from our language $\mathcal{L}$ to a semantic unit $\mathcal{L}_{SU}$, the semantics of $\mathcal{L}$ is defined by $M_{Si} \circ M_S$, i.e. this transformation maps each concept of $\mathcal{L}$ to the semantic domain used by $\mathcal{L}_{SU}$.

Earlier, semantic units were used in the context of operational semantics. In this thesis, we extend the notion of semantic units to denotational semantics. In this case, a semantic unit is a reusable formal representation for a mathematical domain, and for specifying the denotational semantics of a language $\mathcal{L}$ we only need to describe a semantic anchoring from $\mathcal{L}$ into an appropriate semantic unit.

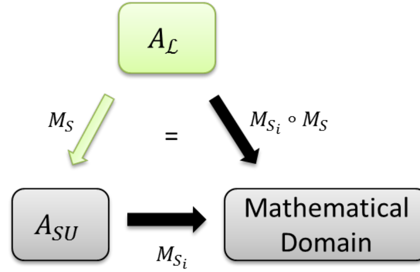


Figure 1: Formalization of behavioral semantics by semantic anchoring. The behavioral semantics of language $\mathcal{L}$ is expressed as a semantic anchoring M_S from its abstract syntax $A_{\mathcal{L}}$ to the abstract syntax A_{SU} of the semantic unit. The semantic mapping M_{S_i} and the mathematical domain are unique for the semantic unit, and developed only once.

Another difference compared to the original semantic anchoring framework is in the language used for specifying the semantic units. Chen et al. [22] use the AsmL language for specifying the syntax and semantics of the semantic units. In this thesis, we develop the abstract syntax of the semantic units using ForSpec, and specify their structural semantics using logic rules. The behaviors described by the Cyber-Physical System (CPS) semantic units are too generic (e.g., trajectories described by differential equations) to be easily expressed in ForSpec (or any other executable specification language for that matter); therefore, we step out from the specification language, and specify them using mathematical logic and calculus. The consequence is that while we can produce the equations belonging to a model, we cannot calculate the trajectories described by them.

Note that this is a more generic problem than choosing a specification language that is powerful enough: calculating the exact trajectory described by some differential equations is generally not possible – only rigorous over-approximations can be calculated (this is the topic of verified integration [14] with applications in particle beam physics and hybrid system verification [118]).

6.2 Primary CPS semantic units

In order to formalize the denotational semantics of the heterogeneous languages used in CPS design, we need heterogeneous semantic units. For instance, the semantics of physical systems modeling languages requires the representation of continuous-time trajectories that are best described by differential algebraic equations. On the other hand, data flow languages used in controller design are often described in terms of discrete-time difference equations. Or, the formalization of high-level controller design languages often require Finite-State Machines (FSMs).

In this section, we define primary semantic units for typical CPS languages. These primary semantic units are then reusable for the specification of CPS languages, as well as for the specification of other semantic units as discussed in the following sections.

6.2.1 Differential algebraic equations

Physical processes are often described using Differential Algebraic Equations (DAEs). A first-order DAE is an equation in the generic form of

$$\mathbf{F}(t, \mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}) = \mathbf{0},$$

where $\mathbf{F}$ is an arbitrary function, $\mathbf{x}$ and $\mathbf{y}$ are vectors of time-dependent variables and t is the independent time variable.

A special class of DAEs is the class of semi-explicit DAEs that can be written in the following form:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{y}, t)\\ \mathbf{0} &= \mathbf{g}(\mathbf{x}, \mathbf{y}, t)\end{aligned}$$

Hence, a semi-explicit DAE can be written as a pair of a differential equation and an algebraic equation.

In the following, we consider only semi-explicit DAEs. We define a language for representing semi-explicit DAEs: first, we define its abstract syntax, then define a semantic mapping that maps each element to the mathematical domain of differential equations. The abstract syntax is formalized by the following algebraic data types:

```
domain DAE
{
  term      ::= cvar + Real + op + {time}.
  op        ::= unaryOp + binaryOp + summa.
  unaryOp   ::= neg + inv.
  binaryOp  ::= add + sub + mul + div.
  equation  ::= eq + diffEq.
  cvar   ::= new (name:String,id:Integer).
  summa  ::= new (name:String,id:Integer).
  neg    ::= new (any term).
  inv    ::= new (any term).
  add    ::= new (any term, any term).
  sub    ::= new (any term, any term).
  mul    ::= new (any term, any term).
  div    ::= new (any term, any term).
  addend ::= new (summa, term).
  eq      ::= new (term, term).
  diffEq ::= new (cvar, term).
}
```

A *term* is a (continuous-time) *variable*, a *real* number, the application of an *operator*, or the independent variable *time*. We define two unary operators (negation, inversion), four binary operators (addition, subtraction, multiplication and division), and an n-ary operator summation. Variables and summations are uniquely identified by their names and integer identifiers, and the *addends* of a sum is represented as relations between the *sum* and its addends. An *equation* is either an algebraic equation *eq* that denotes the equality of the left-hand side and the right-hand side, or a differential equation *diffEq* that denotes the derivative of the left-hand side variable being equal to the right-hand side term.

A trajectory is a real-valued function ν that assigns a real value to each variable in the system at each time $t \in \mathbb{T}$ (typical time domains are the real-time $\mathbb{R}$ and

the hyper-real time ${}^*\mathbb{R}$). Then, the semantics of algebraic equations and differential equations define constraints over the possible trajectories of the system. To develop this idea, first, we have to extend the trajectories from variables to arbitrary terms:

$$
\begin{array}{lll}
\nu^E\colon \mathbb{T} \times \texttt{term} \to \mathbb{R} & & \\
\nu^E(t, \texttt{r}) & \stackrel{def}{=} r & r \in \mathbb{R} \\
\nu^E(t, \texttt{time}) & \stackrel{def}{=} t & \\
\nu^E(t, \texttt{u}) & \stackrel{def}{=} \nu(t, \texttt{u}) & \texttt{u} \text{ is } \texttt{cvar} \\
\nu^E(t, \texttt{neg(u)}) & \stackrel{def}{=} -\nu^E(t, \texttt{u}) & \\
\nu^E(t, \texttt{inv(u)}) & \stackrel{def}{=} 1/\nu^E(t, \texttt{u}) & \\
\nu^E(t, \texttt{add(u,v)}) & \stackrel{def}{=} \nu^E(t, \texttt{u}) + \nu^E(t, \texttt{v}) & \\
\nu^E(t, \texttt{sub(u,v)}) & \stackrel{def}{=} \nu^E(t, \texttt{u}) - \nu^E(t, \texttt{v}) & \\
\nu^E(t, \texttt{mul(u,v)}) & \stackrel{def}{=} \nu^E(t, \texttt{u}) \cdot \nu^E(t, \texttt{v}) & \\
\nu^E(t, \texttt{div(u,v)}) & \stackrel{def}{=} \nu^E(t, \texttt{u}) / \nu^E(t, \texttt{v}) & \\
\nu^E(t, \texttt{summa(i)}) & \stackrel{def}{=} \left(\sum_{\texttt{addend(summa(i),x)}} \nu^E(t, \texttt{x})\right) &
\end{array}
$$

Here, the operators on the right-hand side are the standard operators of real numbers with the usual constraints: e.g., division by zero is undefined.

The semantics of equations are given by the trajectories that satisfy the following formulas:

$$
\begin{array}{ll}
\nu \models \texttt{eq(u,v)} & \text{if } \nu^E(t, \texttt{u}) = \nu^E(t, \texttt{v}) \text{ for all } t \in \mathbb{T} \\
\nu \models \texttt{diffEq(u,v)} & \text{if } d(\nu^E(t, \texttt{u}))/dt = \nu^E(t, \texttt{v}) \text{ for all } t \in \mathbb{T}
\end{array}
$$

Finally, the semantics of an equation system is defined by the trajectories ν that simultaneously satisfy all the equations.

6.2.2 Difference equations

Discrete controllers are often described using Difference Equations (DE). A DE is a recurrence relation that defines the next value of a variable based on its previous values. For example, the following recurrence relation describes the Fibonacci-numbers:

$$
\begin{aligned}
&X[0] = 1, \quad X[1] = 1 \\
&X[k] = X[k-2] + X[k-1], \qquad k \geq 2
\end{aligned}
$$

The abstract syntax of our semantic unit for Difference Equations (DEs) is very similar to that of DAEs:

```
domain DE
{
  term      ::= pre + Real + op + {dtime}.
  op        ::= unaryOp + binaryOp + summa.
  unaryOp   ::= pre + neg + inv.
  binaryOp ::= add + sub + mul + div.
  dvar   ::= new (name:String, id:Integer).
  summa ::= new (name:String, id:Integer).
  pre_dvar ::= pre + dvar.
  pre    ::= new (any pre_dvar).
  neg    ::= new (any term).
  inv    ::= new (any term).
  add    ::= new (any term, any term).
  sub    ::= new (any term, any term).
  mul    ::= new (any term, any term).
  div    ::= new (any term, any term).
  addend ::= new (summa, term).
  equation ::= new (dvar, term).
}
```

A *term* is a previous value of a (discrete-time) *variable*, a *real* number, the application of an *operator*, or *dtime* standing for discrete time. Similar to synchronous languages (e.g., Lustre [43]) we define a *pre* operator that denotes the previous value of a variable. Furthermore, we define two unary operators – negation and inversion –, four

binary operators – addition, subtraction, multiplication and division –, and an n-ary operator summation. The addends of sums are represented as relations between the *sum* and its addend *terms*. Finally, an *equation* denotes a recurrence relation, where the left-hand side is the current value of a discrete-time variable, and the right-hand side refers to previous values of variables.

A trace is a function τ that assigns a value to each variable in the system at each time $t \in \mathbb{N}$. Then, the semantics of equations define constraints over the possible traces of the system. Again, we have to extend traces from variables to terms:

$$
\begin{array}{lll}
\tau^E \colon \mathbb{N} \times \texttt{term} \to \mathbb{R} & & \\
\tau^E(k, \texttt{r}) & \stackrel{\textit{def}}{=} & r \qquad r \in \mathbb{R} \\
\tau^E(k, \texttt{dtime}) & \stackrel{\textit{def}}{=} & k \\
\tau^E(k, \texttt{u}) & \stackrel{\textit{def}}{=} & \tau(k, \texttt{u}) \qquad \texttt{u} \text{ is } \texttt{dvar} \\
\tau^E(k, \texttt{pre(u)}) & \stackrel{\textit{def}}{=} & \tau^E(k-1, \texttt{u}) \\
\tau^E(k, \texttt{neg(u)}) & \stackrel{\textit{def}}{=} & -\tau^E(k, \texttt{u}) \\
\tau^E(k, \texttt{inv(u)}) & \stackrel{\textit{def}}{=} & 1/\tau^E(k, \texttt{u}) \\
\tau^E(k, \texttt{add(u,v)}) & \stackrel{\textit{def}}{=} & \tau^E(k, \texttt{u}) + \tau^E(k, \texttt{v}) \\
\tau^E(k, \texttt{sub(u,v)}) & \stackrel{\textit{def}}{=} & \tau^E(k, \texttt{u}) - \tau^E(k, \texttt{v}) \\
\tau^E(k, \texttt{mul(u,v)}) & \stackrel{\textit{def}}{=} & \tau^E(k, \texttt{u}) \cdot \tau^E(k, \texttt{v}) \\
\tau^E(k, \texttt{div(u,v)}) & \stackrel{\textit{def}}{=} & \tau^E(k, \texttt{u}) / \tau^E(k, \texttt{v}) \\
\tau^E(k, \texttt{summa(i)}) & \stackrel{\textit{def}}{=} & \left(\sum_{\texttt{addend(summa(i),x)}} \tau^E(k, \texttt{x})\right)
\end{array}
$$

Here, the right-hand side operators are the standard operators for real numbers with the usual constraints.

The semantics of equations are given by the traces that satisfy the following formula:

$$\tau \models \texttt{eq(u,v)} \quad \text{if } \tau(k, \texttt{u}) = \tau(k, \texttt{v}) \text{ for all } k \in \mathbb{N}$$

Finally, the semantics of an equation system is defined by the trajectories τ that simultaneously satisfy all the equations.

6.2.3 Finite-state machines

FSMs are tuples of $\langle state, event, transition, init\rangle$ consisting of

- *state*, a finite set of states,
- *event*, a finite set of events,
- a transition relation *transition* $\subseteq$ *state* $\times$ *event* $\times$ *state* that relates pairs of states and events,
- an initial state *init* $\in$ *state*.

We can represent FSMs using the following abstract syntax:

```
domain FSM {
  state ::= new (String).
  event ::= new (String).
  transition ::= new (src:state, ev:event, dst:state).
  init  ::= new (state).
}
```

A trace $s_0 \rightarrow_{e_0} s_1 \rightarrow_{e_1} \ldots$ is a chain of states and events. A FSM *generates* (accepts) a trace if $s_0 = init$ and $(s_i, e_i, s_{i+1}) \in transition$ for any $i \geq 0$. The behavioral semantics of a FSM is the traces it generates (accepts).

6.3 Semantic unit extension and restriction

Next, we discuss the extension and restriction of semantic units. Let *models*$(\mathcal{SU}) = \{M \mid M$ *conforms to* $\mathcal{SU}\}$ denote the set of conforming models for the semantic unit $\mathcal{SU}$. The restriction *restrict*$(\mathcal{SU})$ of a semantic unit $\mathcal{SU}$ is a semantic unit with additional well-formedness constraints on its models. Then, models of the restricted unit are a subset of the models of the original semantic unit: *models*(*restrict*$(\mathcal{SU})) \subseteq$ *models*$(\mathcal{SU})$.

The extension of a semantic unit augments the original language structures by additional elements, i.e. the models of the extended semantic unit is a superset of the models of the original semantic unit: *models*(*extend*$(\mathcal{SU})) \supseteq$ *models*$(\mathcal{SU})$.

Of course, extensions and restrictions can be combined, in which case the relations between the original and the modified semantic units are more complicated.

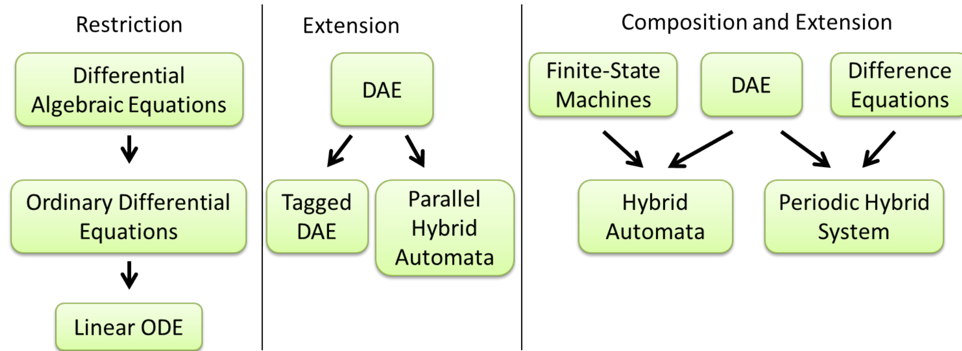


Figure 2: Examples for restriction, extension and composition of semantic units.

In the following, we discuss some semantic units that are restrictions or extensions of the previously introduced primary semantic units. Fig. 2 provides an overview of the discussed semantic units.

6.3.1 Ordinary differential equations

Ordinary Differential Equations (ODEs) are differential algebraic equations without algebraic equation components. The explicit form of a first-order ODE is written as:

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}),$$

where $\mathbf{x}$ are the dependent variables and t is the independent (time) variable.

Clearly, we can restrict (extend with constraints) our primary DAE semantic unit to represent such ODEs:

```
domain ODE extends DAE at "dae.4ml"
{
  conforms no DAE.eq(_,_).
}
```

This conformance rule captures the structural semantics of the ODE domain: a well-formed ODE model is a well-formed DAE model that does not contain any algebraic equations.

6.3.2 Linear ordinary differential equations

An ODE is linear if it can be written as the linear combination of the dependent variables:

$$\mathbf{x}^{(n)} = \sum_{i=0}^{n-1} \mathbf{a_i}(t) \cdot \mathbf{x}^{(i)} + \mathbf{b}(t)$$

where $\mathbf{x}$ are the dependent variables, $\mathbf{x}^{(i)}$ denotes the ith derivative of $\mathbf{x}$ with respect to t, and t is the independent variable.

For first-order ODEs this means the following correspondence:

$$\dot{\mathbf{x}} = \mathbf{a}(t) \cdot \mathbf{x} + \mathbf{b}(t)$$

We can restrict our first-order ODE semantic unit to describe first-order linear ODEs by adding further constraints to its conformance rule:

```
domain linearODE extends ODE at "ode.4ml"
{
  subterm ::= (ODE.term, ODE.term).
  subterm(X,X) :- ODE.diffEq(_,X).
  subterm(X,Z) :-
             subterm(X,neg(Z));
             subterm(X,inv(Z));
             subterm(X,add(Z,_)); subterm(X,add(_,Z));
             subterm(X,sub(Z,_)); subterm(X,sub(_,Z));
             subterm(X,mul(Z,_)); subterm(X,mul(_,Z));
             subterm(X,div(Z,_)); subterm(X,div(_,Z));
             subterm(X,Y), Y:sum, addend(Y,Z).
  nonlinear ::= (ODE.term).
  nonlinear(X) :-
    subterm(X,inv(Z)), subterm(Z,V), V:cvar;
```

```
    subterm(X,mul(S,T)), subterm(S,V1), subterm(T,V2),
                         V1:cvar, V2:cvar;
    subterm(X,div(S,T)), subterm(T,V), V:cvar;
  conforms no nonlinear(_).
}
```

Here, *subterm* extracts subterms of the right-hand side of equations, and *nonlinear* marks all the terms that contain any non-linear subterms. The definition of *nonlinear* consists of rules for the three operators that introduce non-linearity in the system: inv, mul and div. Notice that this means that our semantic unit does not allow equations that could be possibly simplified to linear equations: e.g., $\dot{x} = (x \cdot x) \cdot (\frac{1}{x})$ is an ill-formed equation.

6.3.3 Tagged differential algebraic equations

A different way to extend our equation domains is to associate tags with the variables. This is especially useful for formalizing physical modeling languages, where variables represent physical quantities with units. The following extension of the DAE domain attaches tags to variables:

```
domain TaggedDAE extends DAE at "dae.4ml"
{
  Tag ::= (cvar, String).
}
```

In this case, the tag is a string that can represent arbitrary information. Note that it would be possible to build an algebraic representation for the units; however, we will not pursue this idea in this thesis.

6.3.4 Parallel hybrid automata

Finally, we define a parallel hybrid automata semantic unit that extends the differential algebraic equations semantic unit with parallel running *machines*. Each

machine has two locations – on and off –, which are associated with a set of equations, and an initial location. Furthermore, the transition between the two locations – *switchOn* and *switchOff* – are controlled by events.

A parallel hybrid automaton is a tuple $H = \langle V,$ *machine, event, act, onAct, offAct, label, switchOn, switchOff, initialLoc* $\rangle$ consisting of

- V, a finite set of variables,
- a finite set of *machine*s,
- a finite set of *event*s,
- a set of location-independent activities (equations) *act*,
- a set of location-dependent activities (equation) *onAct* and *offAct*,
- a set of transitions *switchOn* $\subseteq$ *machine* $\times$ *event* that defines the events, upon which a machine transitions from location *off* to location *on*,
- a set of transitions *switchOff* $\subseteq$ *machine* $\times$ *event* that defines the events, upon which a machine transitions from location *on* to location *off*,
- a finite set of initial locations *initialLoc*, one for each machine in the hybrid system.

We formalize this domain as follows:

```
domain ParallelHybridAutomata extends DAE at "dae.4ml"
{
  location ::= { on, off }.
  event     ::= new (String).
  machine   ::= new (id:Integer).
  onAct     ::= new (m:machine, act:eq+diffEq).
  offAct    ::= new (m:machine, act:eq+diffEq).
  initialLoc ::= fun (m:machine => init:location).
  switchOn   ::= new (m:machine, ev:event).
  switchOff  ::= new (m:machine, ev:event).
}
```

Note that the variables V and location-independent activities *act* are inherited from the DAE domain. In particular, V is the continuous-time variables, and *act* consists of the algebraic and differential equations from the DAE domain.

In the following, we describe the semantics of our parallel hybrid automata domain borrowing ideas from the seminal paper by Alur et al. [5]. The behavior of a parallel hybrid automaton is given in terms of the *trajectories* it produces. At any time instant, the state $\sigma \in$ **state** of the system is completely determined by the current locations $L = \{l^1, l^2, \ldots\}$ of the machines (l^i being the location of machine i) and the valuation of variables V. The state can change in two ways: either by a discrete transition between locations, or by the continuous evolution of the variables according to the activities.

We can represent the behavior as a transition system $\langle \Sigma, \rightarrow \rangle$, where Σ denotes the states of the system, and $\rightarrow \subseteq \Sigma^2$ is a binary transition relation between states. We distinguish two types of transitions, $\xrightarrow{t}$ denotes a transition during which the real time advances, and $\xrightarrow{d}$ denotes a discrete jump. A trajectory of the system is a sequence $\sigma_0 \rightarrow \sigma_1 \rightarrow \cdots$ of states, such that each $\langle \sigma_i, \sigma_{i+1} \rangle \subseteq \rightarrow$ for every $i \in \mathbb{N}$. The time advancing transitions $\xrightarrow{t}$ are defined by the location-dependent activities *onAct* and *offAct* and the location-independent activities *act*. The discrete jumps $\xrightarrow{d}$ are defined by the observed events: on the receipt of event e, machine m is fired if there is a transition T of *switchOn* or *switchOff*, such that $T = \langle m, e \rangle$. Upon firing, the system jumps to the next state according to the update function defined in the following.

The *update* function of the parallel automaton is given as a parallel update of the individual machines. In the following, up_m is the update function for machine m:

$$
\begin{aligned}
\textit{update} &: \textbf{state} \times \textbf{event} \rightarrow \textbf{state} \\
up_m &: \textbf{location} \times \textbf{event} \rightarrow \textbf{location} \\
\textit{update}((L_{i-1}, v), e) &= ((up_1(l^1_{i-1}, e), \ldots, up_n(l^n_{i-1}, e)), v) \\
up_m(l, e) &= \begin{cases} \textit{off} & \text{if } l = \textit{on} \wedge \langle m, e \rangle \in \textit{switchOff} \\ \textit{on} & \text{if } l = \textit{off} \wedge \langle m, e \rangle \in \textit{switchOn} \\ l & \text{otherwise} \end{cases}
\end{aligned}
$$

This concludes the definition of the parallel hybrid automata semantic unit.

6.4 Composition of semantic units

Earlier, Chen et al. introduced [21] the composition of semantic units. Based on the semantic units introduced in the previous sections, we can define useful composite semantic units. In this section, we develop two of them: one for representing hybrid differential-difference equations (for describing sampled-data systems), another one for hybrid automata models.

6.4.1 Hybrid differential-difference equations

We can combine the differential algebraic equation and the difference equation domains to create a hybrid equations domain. For this, we need to create syntactic elements for relating continuous-time and discrete-time variables, and define the semantics of these elements.

In our case, we want to describe periodic discrete-time difference equations; therefore, we create the following hybrid domain by adding a timing function for discrete-variables, and sample and zero-order hold operators:

```
domain Hybrid_DAE_DE extends DAE::DAE at "dae.4ml", DE::DE at
    "de.4ml"
{
  // Add hold operator to DAE terms
  DAE.term += hold.
  // Add sample operator to DE terms
  DE.term  += sample.
  // Define timing, sample and hold
  timing ::= fun (DE.dvar ⇒ period:Real, phase:Real).
  sample ::= new (DAE.cvar, period:Real, phase:Real).
  hold   ::= new (DE.dvar).
  // Redefine term and equation for the hybrid domain
  term ::= DAE.term + DE.term + DE.dvar.
  eq   ::= new (term,term).
}
```

The timing relation specifies the timing of discrete-time variables by assigning sampling `period` and initial `phase` to them. The interpretation for periodic discrete-time variables is that they have well-defined values at *real* times $\{p_0 + n \cdot p \mid n \in \mathbb{N}\}$, and everywhere else they are absent. This allows us to migrate the traces τ of DE with the trajectories ν of DAE. We define a hybrid trajectory σ that assigns a real number $\sigma(t, x)$ to each continuous-time variable x and continuous-time t, and assigns a value $\sigma(t, x) \in \mathbb{R} \cup \bot$ to each discrete-time variable x, such that $\sigma(t, x) = \bot$ when x is absent. The semantics of the sample and hold operators are given by the following extension of σ:

$$
\begin{aligned}
&\sigma^E \colon \mathbb{T} \times \texttt{term} \to \mathbb{R} \cup \bot \\
&\sigma^E(t, \mathrm{u}) && \stackrel{\textit{def}}{=} \sigma(t, \mathrm{u}) \\
&\sigma^E(t, \texttt{sample(u,p,ph)}) && \stackrel{\textit{def}}{=} \sigma(t_x, \mathrm{u}) \\
& && \text{where } t_x = \max(\{x = p + n \cdot ph \mid x \leq t \text{ and } n \in \mathbb{N}\}) \\
&\sigma^E(t, \texttt{hold(u)}) && \stackrel{\textit{def}}{=} \sigma(t_x, \mathrm{u}) \\
& && \text{where } t_x = \max(\{x \mid x \leq t \text{ and } \sigma(x, \mathrm{u}) \neq \bot\})
\end{aligned}
$$

6.4.2 Hybrid automata

A well-known model for hybrid systems is the hybrid automaton [5]. Here, we introduce a slightly different definition for hybrid automaton by renaming the labeling functions. A hybrid automaton is a tuple of $\langle V, Q, \textit{flow}, \textit{inv}, \textit{jump} \rangle$ that consists of

- V, a finite set of real-valued *variables*,
- Q, a finite set of *locations* (discrete states),
- a labeling function *flow* that assigns a set of possible activities to each location $q \in Q$, where an activity is a C^∞-function from R^+ to $\mathbb{R}$. We can represent the activities as a set of equations,
- a labeling function *inv* that assigns a set of *invariant* conditions to each location $q \in Q$,

- a labeling function *jump* that assigns a set of jump conditions to pairs of locations.

Note that hybrid automata can be represented as a composition of ODEs and FSMs. The variables V are the continuous-time variables of the ODE, the locations are states of the FSM, *flow* is described by assigning equations to states, *inv* assigns predicates to states, and *jump* assigns equations to transitions (furthermore, in the hybrid automata domain the events corresponding to a transition are only used for distinguishing parallel edges in the graph – the transitions are not triggered by external stimuli).

Therefore, we can define our semantic unit for hybrid automata as follows:

```
domain HybridAutomata extends ODE at "ode.4ml", FSM at "fsm.4ml"
{
  flow ::= new (state, equation).
  inv  ::= new (state, predicate).
  jump ::= new (transition, equation).

  predicate    ::= ODE.eq + less + lessEqual + greater +
   greaterEqual.
  less         ::= new (cvar, term).
  lessEqual    ::= new (cvar, term).
  greater      ::= new (cvar, term).
  greaterEqual ::= new (cvar, term).

  freeEq   :- X is equation, no flow(_,X), no act(_,X).
  freePred :- X is predicate, no inv(_,X), no guard(_,X).
  conforms no freeEq, no freePred.
}
```

Here, the structural semantics requires that each ODE equation is either a flow for some state, or an activity equation for some transition, and each predicate is either an invariant for some state, or a guardian predicate for some transition.

The behavioral semantics of a hybrid automaton is given by the set of runs it accepts as described in [5].

6.5 Conclusion

In this chapter, we discussed the extension of semantic anchoring to denotational specifications. First, we defined several primary semantic units for CPS modeling languages, then we discussed the extension, restriction and composition of these semantic units. We specified the abstract syntax and the structural semantics of these semantic units using the ForSpec language, and defined their behavioral semantics using the tools of calculus and discrete mathematics. The advantage of our approach is that we only need to define these semantic units once, and we can reuse them for the specification of many CPS modeling languages. In the following chapter, we will demonstrate this by discussing several case studies.

CHAPTER 7

CASE STUDIES

In this chapter, we discuss the semantic specification of three Cyber-Physical System (CPS) modeling languages. The first case study will cover a physical modeling language, a bond graph language. The second case study will be a CPS model-integration language called CyPhyML that describes the integration of various physical and controller modeling languages. The third case study will describe the semantics of the MathWorks Stateflow language, a complex language for designing controllers.

While the two first case-studies describe the denotational semantics of the languages, we specify the operational semantics for the MathWorks Stateflow language. Since the operational semantics can be used for calculating the possible trajectories of a system, this opens up the possibility of performing interesting symbolic analysis. In this thesis, we will not pursue this idea; however, it is an important and very useful consequence of the ForSpec specification language.

7.1 Specification of a bond graph language

Bond graph is a multi-domain graphical representation describing the flow of energy in physical systems [62]. Regardless of the domain – electrical, mechanical, thermal, magnetic or hydraulic – the same representation is used for describing the flows. In this section, we define and formalize a bond graph language to demonstrate the formalization of a physical modeling language in our framework.

A bond graph contains nodes and bonds (links) between the nodes (see Fig. 3). Bonds represent the energy exchange between components and are characterized by the power variables: the effort and the flow. The name of these variables is explained by the equation $power = effort \cdot flow$, i.e., their product is the power. Furthermore,

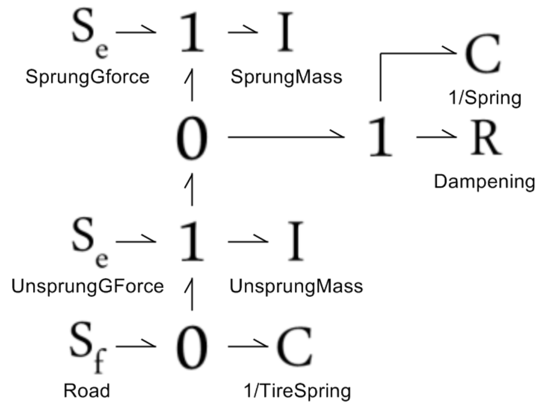


Figure 3: A quarter car suspension model using the bond graph language.

in the basic bond graph, five types of node are distinguished: (i) a dissipative element called resistance **R** having exactly one port, (ii) two storage elements – capacitance **C** and inertia **I** – each having exactly one port, (iii) two source elements – source of effort **Se** and source of flow **Sf** – each having exactly one port, (iv) two transformation elements – transformer **TF** and gyrator **GY** – having exactly two ports, (v) two multi-port topological elements – **0-junction** and **1-junction**.

The description of the nodes provides some hints about their possible meanings (storage, source); however, it is our formal specification that will precisely define their semantics. It is well-known that the behavioral semantics of bond graphs can be described using a set of Differential Algebraic Equations (DAEs) [62]; therefore, we will describe the denotational semantics of our bond graph modeling language by specifying the semantic mapping from the abstract syntax of the language to the DAE semantic unit defined in Chapter 6.

7.1.1 Bond graph structural semantics

The structure of a bond graph represents the energy flows in the physical system. Hence, the structure cannot be arbitrary: it must properly reflect physical reality, which is captured by the structural semantics of the language.

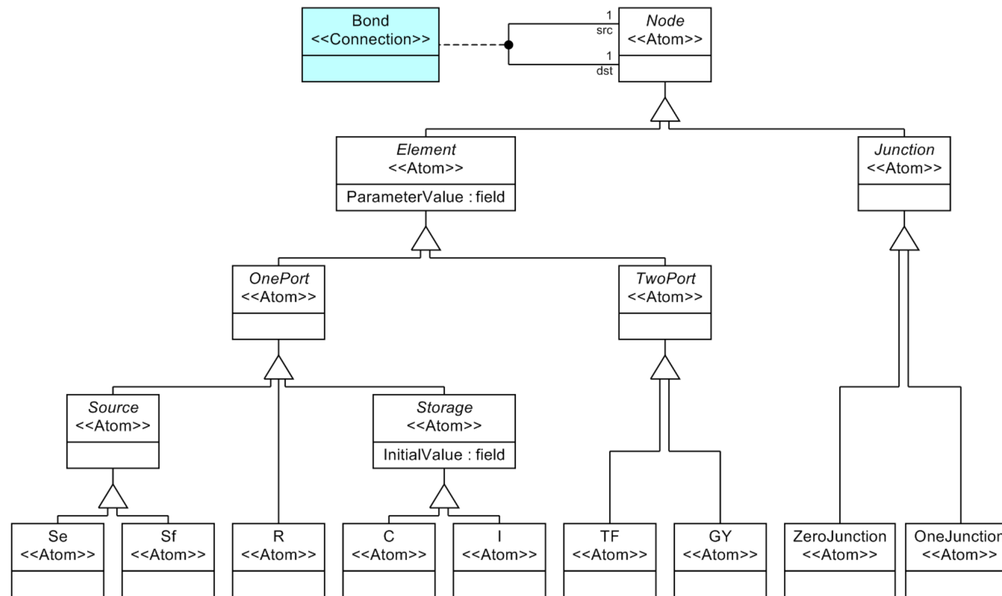


Figure 4: A language for bond graphs as defined in GME.

We have modeled bond graphs in the meta-programmable DSML modeling tool called Generic Modeling Environment (GME) [68]. First, we review the abstract syntax of the language, and then we present its formal structural semantics.

GME meta-model of bond graphs

The GME meta-model of our bond graph language is shown in Fig. 4 with a similar notation to Unified Modeling Language (UML) class diagrams. In brief, the *atom* elements of the meta-model describe classes with attributes, and *connection* elements describe association types. Abstract classes are denoted with italic names, and they are used for creating abstractions of other classes. Triangles denote inheritance, in which case descendants inherit the attributes of their ancestors(s), and whenever an ancestor is allowed to be the source or target of a connection, their descendants can also be used in the same role.

The models of the bond graph domain contain *Nodes* and *Bonds* between them. The multiplicity of the *Bond* element describes that each bond connects exactly one

source *Node* to exactly one destination *Node*. A *Node* is either an *Element* or a *Junction*, where an *Element* is either a *OnePort* or a *TwoPort* element, and *Junction* is either a *ZeroJunction* or a *OneJunction*. Each *Element* has a *ParameterValue* attribute that describes its parameter (e.g. resistance), and is inherited by its children. *OnePort* elements are either *Sources*, *R*esistors or *Storages*. Finally, *Storage* elements have *InitialValue* attributes that describe their initial states.

In GME, the well-formedness rules of the language are expressed with Object Constraint Language (OCL) constraints. For example, we have the following constraint attached to *OnePort* models (not shown in the figure):

```
context: OnePort
inv: self.attachingConnections(Bond)->size = 1
```

This constraint expresses that *OnePort* elements must have exactly one adjacent bond.

ForSpec domain for bond graphs

In order to formalize the semantics of bond graphs, we need to map the GME meta-model to our specification language. In general, we can define a one-to-one mapping from GME meta-models to their ForSpec representation. Each non-abstract class of the meta-model corresponds to a data constructor in the ForSpec domain that can be used to instantiate the class. Furthermore, each class contributes to a union type containing the constructor of the class itself, and the constructors of all their descendant classes. Abstract classes are mapped to union types containing all their descendants. Note that since data constructors and union types cannot have the same name, we distinguish data constructors and corresponding union types by appending $_c$ to the data constructors.

The following types denote the atomic elements of the bond graph domain:

```
Se_c ::= new (name:String, id:Integer).
Sf_c ::= new (name:String, id:Integer).
R_c  ::= new (name:String, id:Integer).
C_c  ::= new (name:String, id:Integer).
```

```
I_c  ::= new (name:String, id:Integer).
TF_c ::= new (name:String, id:Integer).
GY_c ::= new (name:String, id:Integer).
ZeroJunction_c ::= new (name:String, id:Integer).
OneJunction_c  ::= new (name:String, id:Integer).
```

In order to support inheritance, each type has a corresponding abstract union type that contains the type itself and all the inherited subtypes. In our bond graph meta-model, none of the base classes are inherited, so the union types consist of a single element only:

```
Se ::= Se_c.
Sf ::= Sf_c.
R  ::= R_c.
C  ::= C_c.
I  ::= I_c.
TF ::= TF_c.
GY ::= GY_c.
ZeroJunction ::= ZeroJunction_c.
OneJunction  ::= OneJunction_c.
```

The abstract classes of the bond graph meta-model are formalized as follows:

```
Source   ::= Se + Sf.
Storage  ::= C + I.
OnePort  ::= Source + R + Storage.
TwoPort  ::= TF + GY.
Element  ::= OnePort + TwoPort.
Junction ::= ZeroJunction + OneJunction.
Node     ::= Element + Junction.
```

We define a bond as a data type with an identifier and two arguments for its source and destination. Each bond has exactly one source and one destination, which is captured by the following functional relation:

```
Bond_c ::= fun (id:Integer → src:Node, dst:Node).
Bond ::= Bond_c.
```

Finally, attributes are defined as functional types over the corresponding union types. Notice that our encoding correctly represents the inheritance of attributes, and the usage of total functions ensures the correct multiplicity of the attributes.

```
ParameterValue ::= fun (Element ⇒ Real).
InitialValue   ::= fun (Storage ⇒ Real).
```

So far, we formalized the original GME meta-model with an equivalent ForSpec domain that faithfully represents its structure: classes, abstract classes, attributes and connections.

Next, we specify the structural semantics of our bond graph language (this corresponds to the OCL constraints of the meta-model). In order to achieve a concise formalization of the structural and behavioral semantics of bond graphs, we define some derived relations first. We define *src* and *dst* to represent the source and destination of a bond, and *connects* to represent the end-points of a bond:

```
src ::= (Bond,Node).
dst ::= (Bond,Node).
connects ::= (Bond,Node).
src(A,A.src), dst(A,A.dst) :- A is Bond.
connects(A,X) :- src(A,X); dst(A,X).
```

A bond graph is well-formed if it does not contain any invalid nodes:

```
conforms no invalidNode(_).
```

For this, we define a derived data type for deducing invalid nodes:

```
invalidNode ::= (Node).
```

A one-port element is invalid if it is not adjacent to exactly one bond:

```
invalidNode(X) :- X is OnePort, count({Y | connects(Y,X)})!=1.
```

A two-port element is invalid if it is not adjacent to exactly two bonds:

```
invalidNode(X) :- X is TwoPort, count({Y | connects(Y,X)})!=2.
```

Furthermore, a two-port element should have a well-directed energy flow; i.e., each two-port element needs to have exactly one incoming and one outgoing bond:

```
invalidNode(X) :- X is TwoPort, no Bond_c(_,_,X);
                  X is TwoPort, no Bond_c(_,X,_).
```

7.1.2 Bond graph denotational semantics

The behavior of physical systems is usually described by differential algebraic equations, and the interconnections of physical components is described by variable sharing and zero-sum equations [132]. Hence, we define the denotational semantics of our bond graph Domain-Specific Modeling Language (DSML) by specifying the translation from its abstract syntax to the DAEs semantic unit defined in Chapter 6.

We specify the behavioral semantic mapping using a transformation that maps each language concept to a concept in the DAE semantic unit. Each bond $bond_i$ is mapped to two variables – the flow f_i and effort e_i variables indexed by the bond – and each node is mapped to a set of equations constraining the behaviors (possible values over time) of the variables of its bonds. The rules of denotational semantic mapping is described in the following.

We define the semantic functions for bonds and nodes as follows. Bonds are mapped to pairs of continuous-time variables, and nodes are mapped to algebraic equations, differential equations and addends of summations:

```
𝓑 : Bond → cvar,cvar.
𝓝 : Node → eq+diffEq+addend.
```

The semantics of a bond is a pair of continuous-time variables, an effort and a flow variable:

```
𝓑⟦Bond⟧ = (cvar("effort",Bond.id), cvar("flow",Bond.id)).
```

The semantics of a source of effort is the equation $e = p$ that constrains the effort e to the node's parameter value p:

```
𝓝⟦Se⟧ = eq(E, p)
where connects(X,Se), 𝓑⟦X⟧ = (E,_), ParameterValue(Se, p).
```

The semantics of a source of flow is the equation $f = p$ that constraints the flow f to the node's parameter value p:

```
𝒩⟦Sf⟧ = eq(F, p)
where connects(X,Sf), ℬ⟦X⟧ = (_,F), ParameterValue(Sf, p).
```

The semantics of a resistance is the equation $e = r \cdot f$ relating the effort e and flow f, where r is the resistance:

```
𝒩⟦R⟧ = eq(E,mul(r,F))
where connects(X,R), ℬ⟦X⟧ = (E,F), ParameterValue(R, r).
```

The semantics of a capacitance is the differential equation $\dot{e} = \frac{1}{c} \cdot f$ relating the effort e and flow f, where c is the value of the capacitance:

```
𝒩⟦C⟧ = diffEq(E,mul(inv(c),F))
where connects(X,C), ℬ⟦X⟧ = (E,F), ParameterValue(C, c).
```

The semantics of an inductance is the differential equation $\dot{f} = \frac{1}{i} \cdot e$ relating the effort e and flow f, where i is the value of the inductance:

```
𝒩⟦I⟧ = diffEq(F,mul(inv(i),E))
where connects(X,I), ℬ⟦X⟧ = (E,F), ParameterValue(I, i).
```

The semantics of a transformer is two equations ($e_1 = p \cdot e_2$ and $f_2 = p \cdot f_1$) relating the efforts e_1, e_2 and flows f_1, f_2, where p is the value of the transformer:

```
𝒩⟦TF⟧ =
  eq(Ea,mul(p,Eb))
  eq(Fb,mul(p,Fa))
where
  dst(X,TF), src(Y,TF),
  ℬ⟦X⟧ = (Ea,Fa), ℬ⟦Y⟧ = (Eb,Fb),
  ParameterValue(TF, p).
```

The semantics of a gyrator is two equations ($e_1 = p \cdot f_2$ and $e_2 = p \cdot f_1$) relating the efforts e_1, e_2 and flows f_1, f_2, where p is the value of the gyrator:

```
𝒩⟦GY⟧ =
  eq(Ea,mul(p,Fb))
  eq(Eb,mul(p,Fa))
```

```
where
  dst(X,GY), src(Y,GY),
  𝓑⟦X⟧ = (Ea,Fa), 𝓑⟦Y⟧ = (Eb,Fb),
  ParameterValue(GY, p).
```

The semantics of a one-junction is defined by $\forall a,b.(f_a = f_b), \sum e = 0$, where a and b are bonds connected to the junction, and the summation is over all its bonds.

```
// Zero−sum equations
𝓝⟦OneJunction⟧ = eq(summa("OneJunction",OneJunction.id),0).
// Addends of the sum
𝓝⟦OneJunction⟧ = addend(summa("OneJunction",OneJunction.id),E)
where dst(X,OneJunction), 𝓑⟦X⟧ = (E,_).
𝓝⟦OneJunction⟧ =
    addend(summa("OneJunction",OneJunction.id),neg(E))
where src(X,OneJunction), 𝓑⟦X⟧ = (E,_).
// Equality of flows
𝓝⟦OneJunction⟧ = eq(Fa,Fb)
where connects(X,OneJunction), connects(Y,OneJunction), 𝓑⟦X⟧ =
    (_,Fa), 𝓑⟦Y⟧ = (_,Fb).
```

The semantics of a zero-junction is defined by $\forall a,b.(e_a = e_b), \sum f = 0$, where a and b are bonds connected to the junction, and the summation is over all its bonds.

```
// Zero−sum equations
𝓝⟦ZeroJunction⟧ = eq(summa("ZeroJunction",ZeroJunction.id),0).
// Addends of the sum
𝓝⟦ZeroJunction⟧ = addend(summa("ZeroJunction",ZeroJunction.id),F)
where dst(X,ZeroJunction), 𝓑⟦X⟧ = (_,F).
𝓝⟦ZeroJunction⟧ =
    addend(summa("ZeroJunction",ZeroJunction.id),neg(F))
where src(X,ZeroJunction), 𝓑⟦X⟧ = (_,F).
// Equality of efforts
𝓝⟦ZeroJunction⟧ = eq(Ea,Eb)
where connects(X,ZeroJunction), connects(Y,ZeroJunction), 𝓑⟦X⟧ =
    (Ea,_), 𝓑⟦Y⟧ = (Eb,_).
```

7.1.3 Example

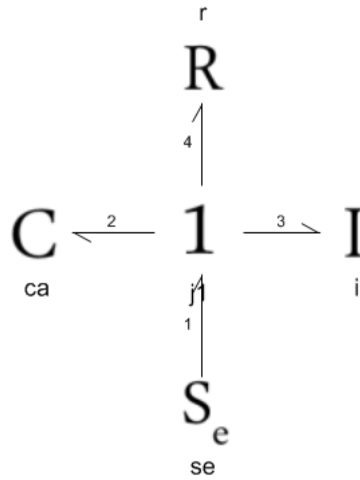
In order to demonstrate the execution of the specifications, consider the bond graph model for an RLC series circuit shown in Fig. 5. The model consists of a resistance, an inductance, a capacitance and a source of effort interconnected with a one-junction.

By loading and executing the structural semantic specifications until reaching a fix-point, we obtain the following facts:

```
model RLC of RLC
{
  ...
  BondGraph.conforms.
  connects(a, j1).
  connects(a, se).
  connects(b, ca).
  connects(b, j1).
  connects(c, i).
  connects(c, j1).
  connects(d, j1).
  connects(d, r).
  dst(a, j1).
  dst(b, ca).
  dst(c, i).
  dst(d, r).
  RLC.conforms.
  src(a, se).
  src(b, j1).
  src(c, j1).
  src(d, j1).
}
```

By observing that `BondGraph.conforms` could be inferred, we know that the RLC model is a well-formed bond graph model.

After executing the denotational semantic mapping, we obtain the semantic model shown in Listing 1 for the circuit.



(a) GME model for the RLC circuit.

```
// Serial RLC circuit
model RLC of BondGraph at "bg.4ml"
{
  j1 is OneJunction_c("j1",1).
  se is Se_c("se",2).
  ca is C_c("ca",3).
  i is I_c("i",4).
  r is R_c("r",5).
  a is Bond_c(1,se,j1).
  b is Bond_c(2,j1,ca).
  c is Bond_c(3,j1,i).
  d is Bond_c(4,j1,r).
  ParameterValue(se,5).
  ParameterValue(ca,1).
  ParameterValue(i,1).
  ParameterValue(r,2).
  InitialValue(ca,1).
  InitialValue(i,0).
}
```

(b) ForSpec model for the RLC circuit.

Figure 5: Bond graph model for an RLC series circuit.

Listing 1: Semantic model for the RLC series circuit

```
model r of DAE at
    "file:///c:/work/documents/thesis/bondgraph/dae.4ml"
{
  addend(summa("OneJunction", 1), cvar("effort", 1)).
  addend(summa("OneJunction", 1), neg(cvar("effort", 2))).
  addend(summa("OneJunction", 1), neg(cvar("effort", 3))).
  addend(summa("OneJunction", 1), neg(cvar("effort", 4))).
  cvar("effort", 1).
  cvar("effort", 2).
  cvar("effort", 3).
  cvar("effort", 4).
  cvar("flow", 1).
  cvar("flow", 2).
  cvar("flow", 3).
  cvar("flow", 4).
  diffEq(cvar("effort", 2), mul(inv(1), cvar("flow", 2))).
  diffEq(cvar("flow", 3), mul(inv(1), cvar("effort", 3))).
  eq(cvar("effort", 1), 5).
  eq(cvar("effort", 4), mul(2, cvar("flow", 4))).
  eq(cvar("flow", 1), cvar("flow", 1)).
  eq(cvar("flow", 1), cvar("flow", 2)).
  eq(cvar("flow", 1), cvar("flow", 3)).
  eq(cvar("flow", 1), cvar("flow", 4)).
  eq(cvar("flow", 2), cvar("flow", 1)).
  eq(cvar("flow", 2), cvar("flow", 2)).
  eq(cvar("flow", 2), cvar("flow", 3)).
  eq(cvar("flow", 2), cvar("flow", 4)).
  eq(cvar("flow", 3), cvar("flow", 1)).
  eq(cvar("flow", 3), cvar("flow", 2)).
  eq(cvar("flow", 3), cvar("flow", 3)).
  eq(cvar("flow", 3), cvar("flow", 4)).
  eq(cvar("flow", 4), cvar("flow", 1)).
  eq(cvar("flow", 4), cvar("flow", 2)).
  eq(cvar("flow", 4), cvar("flow", 3)).
  eq(cvar("flow", 4), cvar("flow", 4)).
  eq(summa("OneJunction", 1), 0).
}
```

7.1.4 Extensibility

Component reusability and extensibility are important concepts for the efficient design of CPS systems. In this section, we demonstrate a possible extension of the bond graph language.

We will extend the language with physical domain-specific elements and with switchable junctions that can be turned on and turned off by (external) controllers. Physical domain-specific elements are labeled with the physical domain they belong to. Such elements can be considered typed elements, where types range over different physical domains. Switchable junctions extend the bond graph language with discrete modes of operations. Such bond graphs are also known as hybrid bond graphs [87].

Physical domain specific elements

A straightforward extension of our bond language is the explicit specification of physical domains for the graph elements (e.g., marking that an inductance stands for an element in the mechanical domain, that is, for a mass). Beside the already discussed well-formedness rules, the structural semantics of the extended language further constrains the allowed connections of nodes.

We can easily augment the abstract syntax of our bond graph language with the necessary marking for the domain types:

```
// Domain is an enumeration of different physical domains.
Domains  ::= {Electrical, Translational, Rotational, Hydraulic,
   Electric}.
// DomainType assigns a domain to one−port and junction elements.
DomainType        ::= fun (OnePort + Junction ⇒ Domains).
// DomainTypeTwoPort assigns two domains to two−port elements: one for each port.
DomainTypeTwoPort ::= fun (TwoPort ⇒ Domains, Domains).
```

Then, we can express the well-formedness of the extended language. An extended bond graph is well-formed if it is a well-formed bond graph (automatically inherited from the BondGraph domain), and all the bonds are valid:

```
conforms no invalidBond(_).
```

A bond is invalid, if it interconnects elements from different domains:

```
invalidBond ::= (Bond).
invalidBond(A) :- A is Bond, DomainType(A.src,X),
    DomainType(A.dst,Y), X != Y.
invalidBond(A) :- A is Bond, DomainType(A.src,X),
    DomainTypeTwoPort(A.dst,Y,_), X != Y.
invalidBond(A) :- A is Bond, DomainType(A.dst,X),
    DomainTypeTwoPort(A.src,_,Y), X != Y.
invalidBond(A) :- A is Bond, DomainTypeTwoPort(A.src,_,X),
    DomainTypeTwoPort(A.dst,Y,_), X != Y.
```

Here, the first rule says that a bond is invalid if it interconnects 1-port or junction elements from different domains, the second and third rules say that a bond is invalid if it interconnects a 1-port or junction element with a 2-port element from a different domain, and the fourth rule says that a bond is invalid if it interconnects 2-port elements from different domains.

Structure of hybrid bond graphs

In this section, we develop the semantics for a hybrid bond graph language that extends our bond graph language with switchable junctions. Fig. 6 shows the GME meta-model for the language, and an example model is shown in Fig. 7.

Switchable junctions are inherited both from the abstract class *SwitchableJunction* and from the original junction classes, *ZeroJunction* and *OneJunction*. Therefore, switchable junctions inherit the attributes of the original junctions, while they are extended with three further attributes: *InitialState*, *OnEvent* and *OffEvent*.

The corresponding ForSpec domain extends the original bond graph domain with data constructors for the new junctions:

```
SwZeroJunction_c ::= new (name:String, id:Integer).
SwOneJunction_c  ::= new (name:String, id:Integer).
SwZeroJunction ::= SwZeroJunction_c.
SwOneJunction  ::= SwOneJunction_c.
```

Also, the union types are updated to reflect the structure of hybrid bond graphs:

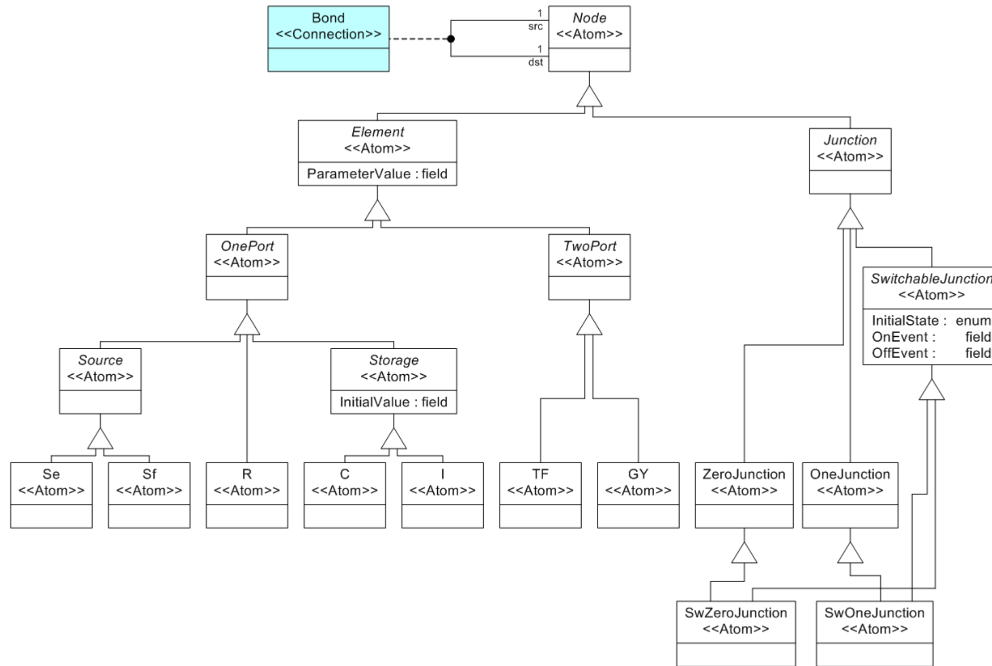


Figure 6: A hybrid bond graph language.

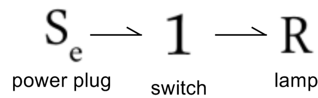


Figure 7: A hybrid bond graph model for a lamp with a switch.

```
SwitchableJunction ::= SwZeroJunction
                     + SwOneJunction.
ZeroJunction ::= ZeroJunction_c + SwZeroJunction.
OneJunction  ::= OneJunction_c + SwOneJunction.
Junctions ::= ZeroJunction + OneJunction
            + SwitchableJunctions.
```

In the GME meta-model, an enumeration is defined with two values – *On* and *Off* – to represent the initial state of switchable junctions. Furthermore, two string attributes – *OnEvent* and *OffEvent* – are defined to store the guards (triggering events) belonging to the switchable junctions. The corresponding attributes in ForSpec are as follows:

```
InitialStateEnum ::= { On, Off }.
InitialState ::= fun (SwitchableJunction =>
                       InitialStateEnum).
OnEvent  ::= fun (SwitchableJunction => String).
OffEvent ::= fun (SwitchableJunction => String).
```

A hybrid bond graph conforms to the hybrid bond graph language if it conforms to the original bond graph language, which is expressed by extending the original bond graph domain.

Behavior of Hybrid Bond Graphs

In order to describe the behavioral semantics of the language, we can use the parallel hybrid automata semantic unit defined in Chapter 6. The behavior of a switchable junction is defined by an automaton with two modes (*On* and *Off*) and two transitions that are triggered by external events (identified by strings). Therefore, each switchable junction is mapped to a machine in the parallel hybrid automata unit.

We specify the denotational semantics of the hybrid bond graph language by extending the denotational semantics of our original bond graph language. This means that the original mappings for the basic bond graph elements result in location-independent activities for the hybrid bond graph language.

Since we extend the original denotational semantic mapping, we reuse the mapping of the original bond graph elements, and we only need to define the mappings for the switchable junctions:

```
SNode [[SwitchableJunction]] = machine(SwitchableJunction.id).
SNode [[SwitchableJunction]] =
   switchOn(machine(SwitchableJunction.id),E)
where OnEvent(SwitchableJunction,E).
SNode [[SwitchableJunction]] =
   switchOff(machine(SwitchableJunction.id),E)
where OffEvent(SwitchableJunction,E).
SNode [[SwitchableJunction]] =
   initialLoc(machine(SwitchableJunction.id),on)
```

```
where InitialState(X,On).
SNode [[SwitchableJunction]] =
    initialLoc(machine(SwitchableJunction.id),off)
where InitialState(X,Off).
```

Second, we need to define the location-dependent activities for the switchable junctions. As long as they are turned on, the behavioral semantics of switchable junctions is exactly the same as the behavior of ordinary junctions, thus the location-dependent activities *onAct* are empty in the turned-on mode. However, when a switchable junction is turned off, it extends the governing differential algebraic equations by forcing the flows across the junction to zero:

```
SNode [[SwitchableJunction]] =
    offAct(machine(SwitchableJunction.id),eq(F,0))
where connects(B,SwitchableJunction), SBond [[B]] => (_,F).
```

Note that this is equivalent to saying that when a junction is turned off, there is no energy flowing through it.

7.1.5 Conclusion

In this chapter, we discussed the semantic specifications of a bond graph language. We specified both its structural and behavioral semantics, as well as presented the extensibility of the language specifications by adding physical domain-specific elements and hybrid behavior.

Since our specifications are executable, model conformance can be automatically computed. This can be used for verifying the well-formedness of a concrete model. Furthermore, by executing the denotational semantic mapping, we can obtain the DAEs describing the behavior of a well-formed model. These equations can be used as an input for other tools, such as simulators and verification tools. For instance, by executing the semantic mapping for a bond graph model, we could use the resulting DAEs to simulate the model in a DAE solver tool (e.g., Modelica [8]).

7.2 Specification of the Cyber-Physical Systems Modeling Language

In this section, we discuss the formalization of the Cyber-Physical Systems Modeling Language (CyPhyML), which is a model integration language for heterogeneous CPS components. CyPhyML is the composition of several sub-languages, such as a language for describing the composition of CPS components, a language for describing design-spaces with multiple choices, and others. In the following, we discuss only the composition sub-language, and by CyPhyML we refer to this language. The GME meta-model [68] of CyPhyML is shown in Fig. 8.

Components are the main building blocks of CyPhyML. Components represent physical or computational elements with ports on their interfaces. Component assemblies are used for building composite structures by composing components and other component assemblies. Component assemblies also facilitate encapsulation and port hiding. There are two types of ports in CyPhyML: acausal power ports for representing physical interaction points, and causal signal ports for representing information flow between components. Both the physical and information flows are interpreted over the continuous time-domain. CyPhyML distinguishes power ports by types, such as electrical power ports, mechanical power ports, hydraulic power ports and thermal power ports.

Formally, a CyPhyML model M is a tuple $M = \langle C, A, P, \textit{contain}, \textit{portOf}, E_P, E_S \rangle$ with the following interpretation:

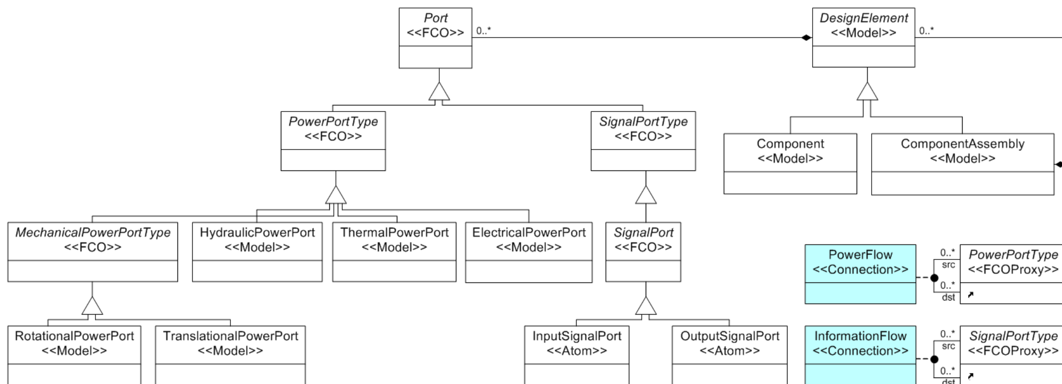


Figure 8: GME meta-model for the composition sub-language of CyPhyML

- C is a set of components,
- A is a set of component assemblies,
- ($D = C \cup A$ is the set of design elements),
- P is the union of the following sets of ports: $P_{rotMech}$ is a set of rotational mechanical power ports, $P_{transMech}$ is a set of translational mechanical power ports, $P_{multibody}$ is a set of multi-body power ports, $P_{hydraulic}$ is a set of hydraulic power ports, $P_{thermal}$ is a set of thermal power ports, $P_{electrical}$ is a set of electrical power ports, P_{in} is a set of continuous-time input signal ports, P_{out} is a set of continuous-time output signal ports. Furthermore, P_P is the union of all the power ports and P_S is the union of all the signal ports,
- $\textit{contain} : D \rightarrow A^*$ is a containment function, whose range is $A^* = A \cup \{\textit{root}\}$, the set of design elements extended with a special root element *root*,
- $\textit{portOf} : P \rightarrow D$ is a port containment function, which uniquely determines the container of a port,
- $E_P \subseteq P_P \times P_P$ is the set of *power flow* connections between power ports,
- $E_S \subseteq P_S \times P_S$ is the set of *information flow* connections between signal ports.

We can formalize this language using the following algebraic data types:

```
// Components, component assemblies and design elements
Component          ::= new (name: String, ..., id:Integer).
ComponentAssembly ::= new (name: String, ..., id:Integer).
DesignElement      ::= Component
                     + ComponentAssembly.
// Components of a component assembly (containment)
ComponentAssemblyToCompositionContainment ::=
    (src:ComponentAssembly, dst:DesignElement).
// Power ports
TranslationalPowerPort ::= new (..., id:Integer).
RotationalPowerPort    ::= new (..., id:Integer).
ThermalPowerPort       ::= new (..., id:Integer).
HydraulicPowerPort     ::= new (..., id:Integer).
ElectricalPowerPort    ::= new (..., id:Integer).
// Signal ports
```

```
InputSignalPort  ::= new (..., id:Integer).
OutputSignalPort ::= new (..., id:Integer).
// Ports of a design element (port containment)
DesignElementToPortContainment ::= new (src:DesignElement,
    dst:Port).
// Union types for ports
Port ::= PowerPortType
       + SignalPortType.
MechanicalPowerPortType ::= TranslationalPowerPort
                          + RotationalPowerPort.
PowerPortType ::= MechanicalPowerPortType
                + ThermalPowerPort
                + HydraulicPowerPort
                + ElectricalPowerPort.
SignalPortType ::= InputSignalPort
                 + OutputSignalPort.
// Connections of power and signal ports
PowerFlow ::=
  new (name:String,src:PowerPortType,dst:PowerPortType,...).
InformationFlow ::=
  new (name:String,src:SignalPortType,dst:SignalPortType,...).
```

7.2.1 Formalization of semantics

Structural Semantics

Next, we formalize the structural semantics of the language. A CyPhyML model is well-formed if it does not contain any dangling ports, distant connections or invalid port connections, hence it `conforms` to the domain:

```
conforms
  no dangling(_),
  no distant(_),
  no invalidPowerFlow(_),
  no invalidInformationFlow(_).
```

For this, we need to define a set of auxiliary rules as discussed next. Dangling ports are ports that are not connected to any other ports:

```
dangling ::= (Port).
dangling(X) :- X is PowerPortType,
  no { P | P is PowerFlow, P.src = X },
  no { P | P is PowerFlow, P.dst = X }.
dangling(X) :- X is SignalPortType,
  no { I | I is InformationFlow, I.src = X },
  no { I | I is InformationFlow, I.dst = X }.
```

A distant connection connects two ports belonging to different components, such that the components have different parents, and neither component is parent of the other one:

```
distant ::= (PowerFlow+InformationFlow).
distant(E) :-
  E is PowerFlow+InformationFlow,
  DesignElementToPortContainment(PX,E.src),
  DesignElementToPortContainment(PY,E.dst),
  PX != PY,
  ComponentAssemblyToCompositionContainment(PPX,PX),
  ComponentAssemblyToCompositionContainment(PPY,PY),
  PPX != PPY, PPX != PY, PX != PPY.
```

A power flow is valid if it connects power ports of the same type:

```
validPowerFlow ::= (PowerFlow).
validPowerFlow(E) :- E is PowerFlow,
  X=E.src, X:TranslationalPowerPort,
  Y=E.dst, Y:TranslationalPowerPort.
validPowerFlow(E) :- E is PowerFlow,
  X=E.src, X:RotationalPowerPort,
  Y=E.dst, Y:RotationalPowerPort.
validPowerFlow(E) :- E is PowerFlow,
  X=E.src, X:ThermalPowerPort,
  Y=E.dst, Y:ThermalPowerPort.
validPowerFlow(E) :- E is PowerFlow,
  X=E.src, X:HydraulicPowerPort,
  Y=E.dst, Y:HydraulicPowerPort.
```

```
validPowerFlow(E) :- E is PowerFlow,
  X=E.src, X:ElectricalPowerPort,
  Y=E.dst, Y:ElectricalPowerPort.
```

If a power flow is not valid, it is invalid:

```
invalidPowerFlow ::= (PowerFlow).
invalidPowerFlow(E) :- E is PowerFlow, no validPowerFlow(E).
```

An information flow is *invalid* if a signal port receives signals from multiple sources, or an output port receives signal from an input port:

```
invalidInformationFlow ::= (InformationFlow).
invalidInformationFlow(X) :-
  X is InformationFlow,
  Y is InformationFlow,
  X.dst = Y.dst, X.src != Y.src.
invalidInformationFlow(E) :-
  E is InformationFlow,
  X = E.src, X:InputSignalPort,
  Y = E.dst, Y:OutputSignalPort.
```

Note that output ports can be connected to output ports.

Denotational Semantics

The denotational semantics of a language is described by a semantic domain and a mapping that maps the syntactic elements of the language to the semantic domain. In this section, we specify a semantic mapping from CyPhyML to the hybrid differential-difference equations semantic unit defined in Chapter 6.

We use the semantic anchoring framework for the denotational semantic specification of CyPhyML as shown in Fig. 9.

CyPhyML distinguishes acausal power ports and causal signal ports. In CyPhyML, each *power port* contributes two variables to the equations, and the denotational semantics of CyPhyML is defined as equations over these variables. *Signal ports* transmit signals with strict causality. Consequently, if we associate a signal

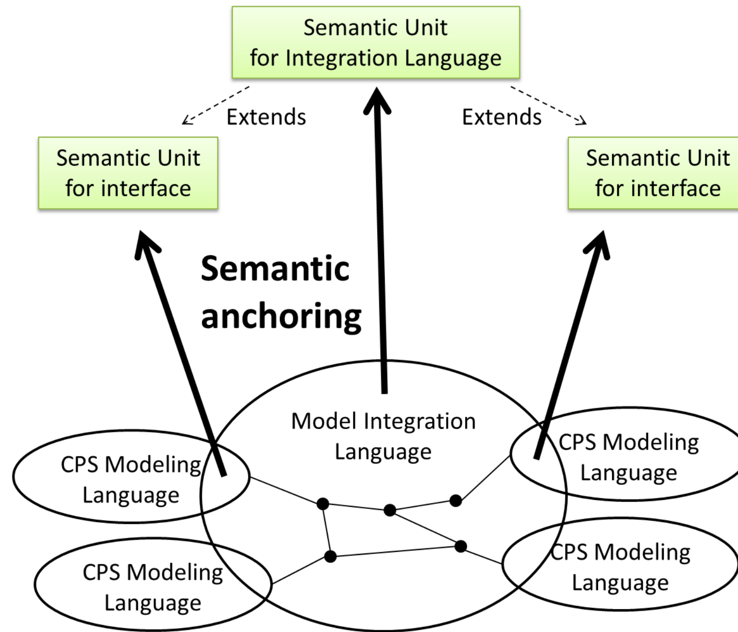


Figure 9: Denotational semantic specification of CyPhyML using semantic anchoring for the semantic interface of the integrated languages and the integration language itself.

variable with each signal port, the variable of a destination port is *enforced* to denote the same value as the variable of the corresponding source port. This relationship is one-way: the value of the variable at the destination port cannot affect the source variable along the connection in question.

The semantic function of power ports maps power ports to pairs of continuous-time variables:

```
𝒫𝒫 : PowerPort → cvar, cvar.
𝒫𝒫⟦CyPhyPowerPort⟧ =
  (cvar("CyPhyML_effort",CyPhyPowerPort.id),
   cvar("CyPhyML_flow",CyPhyPowerPort.id)).
```

The semantic function of signal ports maps signal ports to continuous-time variables:

```
𝒮𝒫 : SignalPort → cvar.
𝒮𝒫⟦CyPhySignalPort⟧ = cvar("CyPhyML_signal",CyPhySignalPort.id).
```

Denotational semantics of power port connections The semantics of power port connections is defined through their transitive closure. Using fixed-point logic, we can easily express the transitive closure of connections as the least fixed point solution for `ConnectedPower`. Informally, `ConnectedPower(x,y)` expresses that power ports `x` and `y` are interconnected through one or more power port connections:

```
ConnectedPower ::= (src:CyPhyPowerPort, dst:CyPhyPowerPort).
ConnectedPower(x,y) :-
  PowerFlow(_,x,y,_,_), x:CyPhyPowerPort, y:CyPhyPowerPort;
  PowerFlow(_,y,x,_,_), x:CyPhyPowerPort, y:CyPhyPowerPort;
  ConnectedPower(x,z), PowerFlow(_,z,y,_,_), y:CyPhyPowerPort;
  ConnectedPower(x,z), PowerFlow(_,y,z,_,_), y:CyPhyPowerPort.
```

In other words, $P_x = \{y \mid ConnectedPower(x, y)\}$ is the set of power ports reachable from power port x.

The behavioral semantics of CyPhyML power port connections is defined by a set of equations generalizing the Kirchoff-equations. Their form is the following:

$$\forall x \in CyPhyPowerPort. \left(isCP(x) \rightarrow \sum_{y \in P_x \wedge isCP(y)} f_y = 0 \right)$$

$$\forall x, y.(ConnectedPower(x, y) \wedge isCP(x) \wedge isCP(y) \rightarrow e_x = e_y)$$

Here, predicate $isCP$ is true for port x iff x is the port of a *component* (that is, not a port of a *component assembly*). We can formalize this the following way:

```
𝒫 : ConnectedPower → eq+out1.DAE.addend.
𝒫⟦ConnectedPower⟧ =
  eq(out1.DAE.summa("CyPhyML_powerflow",flow1.id), 0)
  out1.DAE.addend(summa("CyPhyML_powerflow",flow1.id), flow1)
  out1.DAE.addend(summa("CyPhyML_powerflow",flow1.id), flow2)
  eq(effort1, effort2)
where
  x = ConnectedPower.src, y = ConnectedPower.dst,
  DesignElementToPortContainment(cx,x), cx:Component,
  DesignElementToPortContainment(cy,y), cy:Component,
  𝒫𝒫⟦x⟧ = (effort1,flow1),
```

```
𝒫𝒫⟦y⟧ = (effort2,flow2).
```

Semantics of Signal Port Connections A signal connection path is a directed path along signal connections. We can use fixed-point logic to express the transitive closure of signal connections (`ConnectedSignal`) to find the signal connection paths. Informally, `ConnectedSignal(x,y)` expresses that there is a signal path (chain of connections) from signal port `x` to signal port `y`.

```
ConnectedSignal ::= (CyPhySignalPort,CyPhySignalPort).
ConnectedSignal(x,y) :-
  InformationFlow(_,x,y,_,_),
  x:CyPhySignalPort,
  y:CyPhySignalPort.
ConnectedSignal(x,y) :-
  ConnectedSignal(x,z),
  InformationFlow(_,z,y,_,_),
  y:CyPhySignalPort.
```

In other words, $P_x = \{y \mid ConnectedSignal(x,y)\}$ is the set of signal ports reachable from signal port x.

A `SignalConnection` is a `connectedSignal` such that its end-points are signal ports of components (therefore leaving out any signal ports that are ports of component assemblies).

```
SignalConnection ::= (src:CyPhySignalPort,dst:CyPhySignalPort).
SignalConnection(x,y) :-
  ConnectedSignal(x,y),
  DesignElementToPortContainment(cx,x), cx:Component,
  DesignElementToPortContainment(cy,y), cy:Component.
```

The behavioral semantics of CyPhy signal connections is defined as variable assignment. The values of the variables associated with the source and the destination of the signal connection are equal.

$$\forall x, y.(SignalConnection(x,y) \rightarrow s_y = s_x)$$

```
𝒮 : SignalConnection → eq.
𝒮⟦SignalConnection⟧ =
  eq(𝒮𝒫⟦SignalConnection.dst⟧, 𝒮𝒫⟦SignalConnection.src⟧).
```

7.2.2 Formalization of language integration

So far, we formally defined the semantics of the compositional elements of CyPhyML but we have not specified how components are integrated into CyPhyML. In this section, we develop the semantics of the integration of external languages: a bond graph language, the Modelica language and the Embedded Systems Modeling Language (ESMoL). Note that in the future we can easily add other languages to the list following the same steps as presented here.

Bond Graph is a multi-domain graphical representation for describing the flow of energy in physical systems [62]. In Section 7.1, we introduced a bond graph language along with its formal semantics. Here, we consider an extended bond graph language that defines power ports in addition: these are ports, through which a bond graph component interacts with its environment. Each power port is adjacent to exactly one bond; therefore, a power port represents a pair of power variables: the power variables of its (unique) bond. The bond graph language we consider here also contains output signal ports for measuring efforts and flows at bond graph junctions, and modulated bond graph elements that are controlled by input signals fed to the bond graph through input signal ports. Note that the effort and flow variables of the bond graph language are different from the effort and flow variables of CyPhyML: they denote different entities in different physical domains. The semantics of the languages formalize these differences precisely.

Modelica is an equation-based object-oriented language [35] used for systems modeling and simulation. Modelica supports component-based development through its *model* and *connector* concepts. Models are components with internal behavior and a set of ports called connectors. Models are interconnected by connecting their connector interfaces. A connector is a set of variables (input, output, acausal flow or potential, etc.) and the connection of connectors define relations over their variables.

In the following, we discuss the integration of a restricted set of Modelica models in CyPhyML: we consider models that contain connectors that consists of either exactly one input/output variable, or a pair of potential and flow variables.

The **Embedded Systems Modeling Language** (ESMoL [100]) is a language and tool-suite for modeling and implementing computational systems and hardware platforms. ESMoL consists of several sub-languages for defining platform and software architectures, describing the deployment of software on hardware, and specifying the scheduling of execution. In the following, by ESMoL we refer to the dataflow sub-language of ESMoL that is used for modeling discrete controllers. This sub-language is based on a periodic time-triggered execution semantics, and its models expose periodic discrete-time signal ports on their interfaces.

Integration of structure

The role of CyPhyML in the integration process is to establish meaningful and valid connections between heterogeneous models. Component integration is an error-prone task because of the slight differences between different languages. For instance, during the formalization we found the following discrepancies:

1. power ports have different meaning in different modeling languages,
2. even if the semantics is the same, there are differences in the naming conventions,
3. connecting the signals of ESMoL to the signals of CyPhyML needs conversion between discrete-time and continuous-time signals.

In order to formalize the integration of external languages, we extend CyPhyML with the *semantic interfaces* of these languages. Hence, we need language elements for representing models of these heterogeneous languages, their port structures, and the port mapping between the ports and the corresponding CyPhyML ports.

We formalize the models and their containment in CyPhyML as follows:

```
BondGraphModel  ::= new (URI:String, id:Integer).
ModelicaModel   ::= new (URI:String, id:Integer).
ESMoLModel ::= new (URI:String, id:Integer, sampleTime:Real).
Model ::= BondGraphModel + ModelicaModel + ESMoLModel.
// A relation describing the containment of bond graph models in CyPhyML components
ComponentToBondGraphContainment ::= new (Component =>
    BondGraphModel).
...
```

Note the sampleTime field of *ESMoLModel*: since ESMoL models are periodic discrete-time systems, we need real values describing their period in the continuous-time world. The interface ports and port mappings are the following:

```
// Bond graph power ports (and similarly for the other languages)
BGPowerPort ::= MechanicalDPort + MechanicalRPort + ...
...
// Port mappings for bond graph power ports (and similarly for other languages)
BGPowerPortMap  ::= (src:BGPowerPort,dst:CyPhyPowerPort).
...
// All the power ports in CyPhyML and the integrated languages:
PowerPort ::= CyPhyPowerPort + BGPowerPort + ModelicaPowerPort.
// All the signal ports in CyPhyML and the integrated languages:
SignalPort ::= ElectricalSignalPort
             + BGSignalPort
             + ModelicaSignalPort
             + ESMoLSignalPort.
// List of all ports:
AllPort ::= PowerPort + SignalPort.

// Mapping from model ports to CyPhyML ports
PortMap ::= BGPowerPortMap
          + BGSignalPortMap
          + ModelicaPowerPortMap
          + ModelicaSignalPortMap
          + SignalFlowSignalPortMap.
```

An integrated model (that is, CyPhyML model integrated with other models) is well-formed if it conforms to the original CyPhyML domain, and its port mappings are valid:

```
conforms no invalidPortMapping.
```

A port mapping is invalid if it connects incompatible ports, or the interconnected ports are not part of the same CyPhyML component:

```
invalidPortMapping :- M is PortMap, no compatible(M).
invalidPortMapping :-
  M is BGPowerPortMap,
  BondGraphToPortContainment(BondGraph,M.src),
  DesignElementToPortContainment(CyPhyComponent,M.dst),
  no ComponentToBondGraphContainment(CyPhyComponent,BondGraph).
...
// Compatible denotes that port mapping M is valid (i.e., the corresponding ports are compatible)
compatible ::= (PortMap).
compatible(M) :- M is BGPowerPortMap(X,Y), X:MechanicalRPort,
    Y:RotationalPowerPort.
...
```

Bond Graph integration The semantics of bond graph power ports are explained by mapping to pairs of continuous-time variables:

```
𝓑𝓖𝓟𝓟 : BGPowerPort → cvar, cvar.
𝓑𝓖𝓟𝓟⟦BGPowerPort⟧ =
   (cvar("BondGraph_effort",BGPowerPort.id),
    cvar("BondGraph_flow",BGPowerPort.id)).
```

The semantics of bond graph signal ports is explained by mapping to continuous-time variables:

```
𝓑𝓖𝓢𝓟 : BGSignalPort → cvar.
𝓑𝓖𝓢𝓟⟦BGSignalPort⟧ = cvar("BondGraph_signal",BGSignalPort.id).
```

The behavioral semantics of bond graph power port mappings for the hydraulic and thermal domains is the equality of the associated port variables. We can formalize this with the following rules:

```
𝓑𝓖𝓟 : BGPowerPortMap → eq+diffEq.
𝓑𝓖𝓟⟦BGPowerPortMap⟧ =
  eq(cyphyEffort, bgEffort)
  eq(cyphyFlow, bgFlow)
where
  bgPort = BGPowerPortMap.src,
  cyphyPort = BGPowerPortMap.dst,
  bgPort : HydraulicPort + ThermalPort,
  𝓟𝓟⟦cyphyPort⟧ = (cyphyEffort, cyphyFlow),
  𝓑𝓖𝓟𝓟⟦bgPort⟧ = (bgEffort, bgFlow).
```

In mechanical translational domain, the effort of CyPhyML power ports denote absolute position and the flow denotes force, whereas for bond graphs the effort is force, and the flow is velocity. In mechanical rotational domain, the effort of CyPhyML power ports denote absolute rotation angle and the flow denotes torque, whereas for bond graphs the effort is torque and the flow is angular velocity. Their interconnection in CyPhyML is formalized by the following equations:

```
𝓑𝓖𝓟⟦BGPowerPortMap⟧ =
  diffEq(cyphyEffort, bgFlow)
  eq(bgEffort, cyphyFlow)
where
  bgPort = BGPowerPortMap.src,
  cyphyPort = BGPowerPortMap.dst,
  bgPort : MechanicalDPort + MechanicalRPort,
  𝓟𝓟⟦cyphyPort⟧ = (cyphyEffort, cyphyFlow),
  𝓑𝓖𝓟𝓟⟦bgPort⟧ = (bgEffort, bgFlow).
```

For the electrical domain, bond graph electrical power ports denote a pair of physical terminals (electrical pins), while in the CyPhyML language they denote single electrical pins. In both cases, the flows (the currents) through the pins are the same; however, there are differences in the interpretation of the voltages. In the bond graph case, the effort variable belonging to the electrical power port denotes the difference of the voltages between the two electrical pins. In the CyPhyML case, the effort variable denotes absolute voltage with respect to an absolute zero voltage. The semantics of electrical power port mapping is the equality of the flows and efforts,

which means that the negative terminal of the bond graph electrical power port is automatically grounded to zero voltage:

```
𝓑𝓖𝓟⟦BGPowerPortMap⟧ =
  eq(bgFlow, cyphyFlow)
  eq(bgEffort, cyphyEffort)
where
  bgPort = BGPowerPortMap.src,
  cyphyPort = BGPowerPortMap.dst,
  bgPort : ElectricalPort,
  𝓟𝓟⟦cyphyPort⟧ = (cyphyEffort, cyphyFlow),
  𝓑𝓖𝓟𝓟⟦bgPort⟧ = (bgEffort, bgFlow).
```

Finally, the denotation of bond graph and CyPhyML signal port mapping is equality of the interconnected port variables:

```
𝓑𝓖𝓢 : BGSignalPortMap → eq.
𝓑𝓖𝓢⟦BGSignalPortMap⟧ =
  eq(𝓑𝓖𝓢𝓟⟦BGSignalPortMap.src⟧, 𝓢𝓟⟦BGSignalPortMap.dst⟧).
```

Modelica integration The semantics of Modelica power ports (that is, a connector with a flow and a potential variable) are explained by mapping to pairs of continuous-time variables:

```
𝓜𝓟𝓟 : ModelicaPowerPort → cvar,cvar.
𝓜𝓟𝓟⟦ModelicaPowerPort⟧ =
  (cvar("Modelica_potential",ModelicaPowerPort.id),
   cvar("Modelica_flow",ModelicaPowerPort.id)).
```

The semantics of Modelica signal ports (that is, a connector with an input or output variable) is explained by mapping to continuous-time variables:

```
𝓜𝓢𝓟 : ModelicaSignalPort → cvar.
𝓜𝓢𝓟⟦ModelicaSignalPort⟧ =
   cvar("Modelica_signal",ModelicaSignalPort.id).
```

The semantics of Modelica and CyPhyML power port mappings is equality of the power variables. Formally,

```
𝓜𝓟 : ModelicaPowerPortMap → eq.
𝓜𝓟 ⟦ModelicaPowerPortMap⟧ =
  eq(cyphyEffort, modelicaEffort)
  eq(cyphyFlow, modelicaFlow)
where
  modelicaPort = ModelicaPowerPortMap.src,
  cyphyPort = ModelicaPowerPortMap.dst,
  𝓟𝓟 ⟦cyphyPort⟧ = (cyphyEffort, cyphyFlow),
  𝓜𝓟𝓟 ⟦modelicaPort⟧ = (modelicaEffort, modelicaFlow).
```

The semantics of Modelica and CyPhyML signal port mappings is equality of the signal variables.

```
𝓜𝓢 : ModelicaSignalPortMap → eq.
𝓜𝓢 ⟦ModelicaSignalPortMap⟧ =
  eq(𝓜𝓢𝓟 ⟦ModelicaSignalPortMap.src⟧,
     𝓢𝓟 ⟦ModelicaSignalPortMap.dst⟧).
```

SignalFlow integration The semantics of ESMoL signal ports is explained by mapping to discrete-time variables, and the periodicity of the discrete variable is determined by the sample time of its container block.

```
𝓔𝓢𝓟 : ESMoLSignalPort → dvar, timing.
𝓔𝓢𝓟 ⟦ESMoLSignalPort⟧ = (Dvar, timing(Dvar, container.SampleTime,
   0))
where
  Dvar = dvar("ESMoL_signal", ESMoLSignalPort.id),
  IOSignal2InPort(_,ESMoLSignalPort,inport,_,_),
  BlockToSF_PortContainment(container,inport).
𝓔𝓢𝓟 ⟦ESMoLSignalPort⟧ = (Dvar, timing(Dvar, container.SampleTime,
   0))
where
  Dvar = dvar("ESMoL_signal", ESMoLSignalPort.id),
  OutPort2IOSignal(_,outport,ESMoLSignalPort,_,_),
  BlockToSF_PortContainment(container,outport).
```

While signal ports in signal-flow have discrete-time semantics, signal ports in CyPhyML are continuous-time. Thus, signal-flow output signals are integrated into

CyPhyML by means of the hold operator.

$$\forall x, y.(SignalFlowSignalPortMap(x, y) \rightarrow e_y = hold(e_x))$$

```
ℰ𝒮 : SignalFlowSignalPortMap → eq.
ℰ𝒮⟦SignalFlowSignalPortMap⟧ = eq(cyphySignal,
    hold(signalflowSignal))
where
  signalflowPort = SignalFlowSignalPortMap.src,
  cyphyPort = SignalFlowSignalPortMap.dst,
  signalflowPort : OutSignal,
  𝒮𝒫⟦cyphyPort⟧ = cyphySignal,
  ℰ𝒮𝒫⟦signalflowPort⟧ = (signalflowSignal,_).
```

For the opposite direction, we can use the sampling operator.

$$\forall x, y.(SignalFlorSignalPortMap(x, y) \rightarrow s_x = sample(s_y, rate, phase))$$

The sampling rate and phase of the sampling function is calculated from the timing of the discrete variable corresponding to the ESMoL port:

```
ℰ𝒮⟦SignalFlowSignalPortMap⟧ = eq(signalflowSignal,
    sample(cyphySignal,samp.period,samp.phase))
where
  signalflowPort = SignalFlowSignalPortMap.src,
  cyphyPort = SignalFlowSignalPortMap.dst,
  signalflowPort : InSignal,
  𝒮𝒫⟦cyphyPort⟧ = cyphySignal,
  ℰ𝒮𝒫⟦signalflowPort⟧ = (signalflowSignal,samp).
```

Power Port Units Next, we define the physical units for each of the physical power ports. The Units enumeration contains all the supported physical units:

```
Units ::= {
  "V",      // Voltage
  "A",      // Ampere
  "m",      // meter
```

```
  "N",      // Newton
  "N.m",    // Newton-meter
  "m/s",    // meter/second
  "rad",    // radian
  "rad/s",// radian/second
  "kg/s",  // kilogram/second
  "Pa",     // Pascal
  "K",      // Kelvin
  "W",      // Watt
  "NA",     // Not available
  "J/kg",  // Joule/kilogram
  "Pa,J/kg",
  "kg/s,W"  // Modelica FlowPort
  }.
```

PortUnit assigns two units to each power port: one to its effort variable, and one to its flow variable:

```
PortUnit ::= [port:PowerPort ⇒ effort:Units, flow:Units].
PortUnit(x,"V","A") :- x is ElectricalPowerPort;
                       x is ElectricalPin;
                       x is ElectricalPort.
PortUnit(x,"m","N") :- x is TranslationalPowerPort;
                       x is TranslationalFlange.
PortUnit(x,"N","m/s") :- x is MechanicalDPort.
PortUnit(x,"rad","N.m") :- x is RotationalPowerPort;
                           x is RotationalFlange.
PortUnit(x,"N.m","rad/s") :- x is MechanicalRPort.
PortUnit(x,"kg/s","Pa") :- x is HydraulicPowerPort;
                           x is FluidPort;
                           x is HydraulicPort.
PortUnit(x,"K","W") :- x is ThermalPowerPort;
                       x is HeatPort;
                       x is ThermalPort.
PortUnit(x,"NA","NA") :- x is MultibodyFramePowerPort.
PortUnit(x,"Pa,J/kg","kg/s,W") :- x is FlowPort.
```

It is an interesting future work to use these units to verify the consistency of the language; in particular the consistency of the port mappings, where different modeling languages may use different units.

7.2.3 Conclusion

In this chapter, we discussed how the ForSpec language can be used for specifying both the structural and the denotational behavioral semantics of a CPS integration language. Our approach has two advantages: (i) we used an executable formal specification language, which lends itself to model conformance checking, model checking and model synthesis; (ii) both the structural and behavioral specifications are written using the same logic-based language, therefore both can be used for deductive reasoning: in particular, structure-based proofs about behaviors become feasible. It remains a future work to leverage the specifications for performing such symbolic formal analysis.

7.3 Specification of the Stateflow language

7.3.1 Introduction

Model-based engineering has been successfully applied to tackle the increasing complexity of embedded software by raising the level of abstraction. By now, it is common practice to use high-level modeling languages – such as the statechart formalism or one of its variants – to model controller systems.

In this chapter, we develop executable formal semantic specifications for one of the most prominent modeling language for embedded systems, the MathWorks' Stateflow language [77]. Stateflow is a complex language integrated into the Matlab Simulink environment combining hierarchical state diagrams (similar to Harel's statecharts [46]) with flowcharts, truth tables, graphical functions, a built-in action language, and external Simulink, Matlab and C functions. The complexity of the language is hallmarked by its 896-page long user's guide.

As Stateflow is one of the most widely used statechart variant in the industry, many authors have developed the formal specification of its semantics. We base our semantic specifications on these works, but provide additional formalization for lacking features, and fix some errors. Our contributions are the following:

- **Static (structural) semantics**: formalization of the well-formedness rules of Stateflow.
- **Action and condition languages**: previous attempts abstracted away the action language. We provide the formalization of the action and condition languages.
- **Name resolution for identifiers**: formalization of the name resolution in the action and condition languages.
- **Early return logic**: formalization of the early return logic rules for transition, condition, entry, during and exit actions.

- **History junctions**: formalization of history junctions.
- **On actions**: formalization of on actions besides entry, during and exit actions.
- **Ordering of outer and inner transitions**: proper formalization of inner and outer transitions.
- **Super-step semantics**: we formalized both the single-step and super-step semantics of Stateflow.
- **Default transitions**: we fixed an error in previous formalizations with regards to the default transitions in an AND composition.

7.3.2 Related work

Hamon [45] formalizes the structural operational semantics for a subset of the Stateflow language. The restrictions are the following: local event broadcasts can only be sent to parallel states; loops are forbidden in event broadcasting; history junctions, the action language and name resolution, as well as the early return logic are missing from the specification. The authors use the specifications to develop an interpreter for translating Stateflow models to the SAL language, which is the input language for SRI's formal tools.

Caspi et al. [112] describe a translation from a subset of Stateflow to Lustre. They define a safe subset of Stateflow that can be mapped to Lustre, and provide an informal presentation of the translation process.

Hamon and Rushby [44] formalize the denotational semantics for a larger subset of Stateflow, still missing several of the above-mentioned features. They claim that the advantage of formalizing the denotational semantics is that the specifications define a compiler for Stateflow, which is much easier to maintain compared to the interpreter based on the operational semantics. Their implementation is based on the OCaml language, and by adding pretty-printers, they can produce SAL, C and OCaml code from Stateflow models. We found a bug in this work with regards to the ordering of outer transitions, during actions and inner transitions.

Whalen [131] describes a more complete formalization of Stateflow. Besides the structural operational semantics (SOS) of Stateflow, Whalen also describes the SOS of UML Statecharts and Rhapsody, another two prominent statechart variants. The specifications are parametric, and highlight the commonalities and differences between these variants. Our operational semantics is based on Whalen's work, but we extend it with many additional features. An important basic feature of Stateflow missing from Whalen's specifications is the notion of inner transitions, which we added in this work. Furthermore, we also address history junctions, the action language, name resolution and the early return logic.

7.3.3 Semantics by example

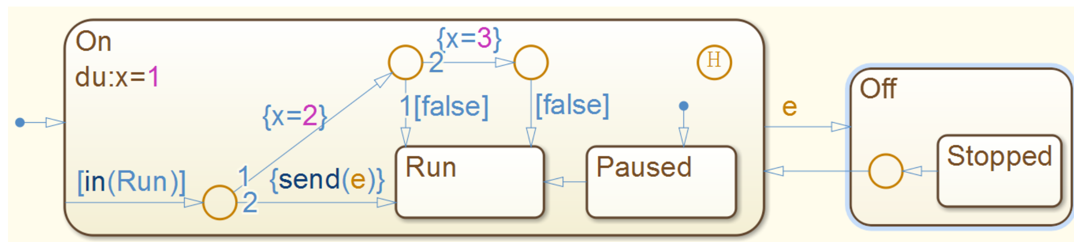


Figure 10: A tricky Stateflow example.

In this section, we demonstrate the execution of Stateflow through an example. For a complete discussion to the topic, we refer the reader to the user guide [77]. We designed the example with the specific aim to show several tricky features of Stateflow that were either not addressed in earlier works, or were incorrectly formalized. Fig. 10 shows the example. The chart is an exclusive composition of two states – *On* and *Off* – with two unguarded transitions inbetween. The transition from *On* to *Off* is triggered by local event *e*, while the transition from *Off* to *On* is always enabled.

The *On* state contains two substates – *Run* and *Paused* –, and its default transition leads to *Paused*. Furthermore, *On* has a during action setting data variable *x* to one, a history junction, and an inner flowchart (connective junctions interconnected through transitions) guarded by *[in(Run)]*, i.e., only executed if substate *Run* is active. The *Off* state contains a single *Stopped* state.

We describe the execution of the chart through four steps. The title for each step emphasizes the important features explained at that step.

Step 1. Initialization: Initially, the chart is inactive and is triggered by the execution environment. This activates the chart, and its default transition is executed. The default transition leads to the *On* state, which in turns executes the default transition of the *On* state, therefore the *On* and *Paused* states become active. No more actions are performed at this point, and the chart goes to sleep.

Step 2. Outer, inner transitions and during actions: On the next trigger of the chart, the chart is already initialized, therefore it executes its active child: state *On*. The outer transition of *On* is executed, but it fails because event e was not triggered. Therefore, the execution of *On*'s during action follows, which sets x to one. Next, the inner transition is evaluated, but state *Run* is not active, therefore it fails. This leads to the execution of *On*'s active child, state *Paused*, which has an enabled (since it has no guards or trigger events) outer transition leading to state *Run*. The transition is taken by exiting *Paused* and entering *Run*, after which the chart goes to sleep.

Step 3. Flowchart execution, backtracking, event broadcast: Similarly, on the third trigger, state *On* executes, but its outer transition fails. Next, the during action successfully sets x to one. This time, the inner transition is taken, because state *Run* is active: first the upper branch is taken (order 1), where the value of x is set to two. Probing the next transition fails (because of guard *[false]*), thus after backtracking the next transition is taken, and variable x is set to three. Again, the outgoing transition fails, and backtracking leads back to the lower branch. Here, event e is broadcast, which immediately executes the chart from the top level:

- Since this time the outer transition is triggered, the active states are exited (first *Run*, then *On*) and state *Off* entered. Even though *Off* does not have a default transition, a special rule states that if a state has only one substate, it is automatically activated. Therefore, *Stopped* is entered as well. This completes the execution of the broadcast event.

At this point, the execution returns to the lower branch in the inner transition of *On*. Since *On* is inactive by now, the early return logic prescribes that the rest of the actions are not performed. The chart goes to sleep.

Step 4. Inter-level transition path, history junction: On the fourth trigger, the inter-level transition path is taking the chart from state *Stopped* to state *On* through the connective junction in *Off*. Since the parent of the transition path is the chart, state *Stopped* and *Off* are exited, and state *On* is entered, respectively. Because of *On*'s history junction, its last active state is entered, that is state *Run*. The chart goes to sleep.

7.3.4 Informal semantics

A Stateflow chart is a reactive model, that is externally triggered by the environment: either by an event, or the passage of time. During the reaction, the chart executes and produces some output. Stateflow is composed of two main parts:

- **Diagram language:** structurally, Stateflow is a hierarchical diagram – a forest –, where the roots are charts, the internal nodes are states or graphical functions, and the leafs are either states or junctions. The nodes of the diagram are interconnected through transitions that are of three types: default transitions, inner transitions and outer (outgoing) transitions.
- **Action language:** Stateflow has an action language that defines operations over the chart's input, output and local data variables. The input variables obtain their values from the environment, the local variables are used by the chart for internal computations, and the output variables store return values that the environment can access.

At any point, the status of the system is completely determined by its active states and the valuation of its local variables.

Diagram language

The diagram language has the following concepts:

- **Charts** are top-level containers that are activated by the environment. A chart has a set of default transitions that are executed if the chart has no active sub-states upon its activation – otherwise, its active sub-states are executed. Charts are either parallel (AND) or exclusive (OR) compositions. An exclusive chart may have at most one active sub-state. In contrast, if a parallel chart has at least one active sub-state, all its sub-states must be active.

- **States** form the backbone of Stateflow. States have a set of default transitions, a set of inner transitions and a set of outer transitions. The default transitions of a state are executed upon the entry of the state. Otherwise, upon the execution of an active state, its outer and inner transitions are executed. States have entry, during, exit and on actions, which are expressions in the action language. States have local variable definitions that can shadow variables from upper levels in the hierarchy, whenever an action expression is evaluated in the context of the state. Similar to charts, states are either parallel or exclusive compositions.

- **Subcharts:** to facilitate compact models Stateflow allows the creation of subcharts from states. This serves only visualization purposes and the semantics of the states remains unchanged.

- **Graphical functions** are Stateflow diagrams that can be called from the action language. Graphical functions have input and output variables, a set of connective junctions and transitions that define the calculations performed by the function.

- **Connective junctions** are used for defining decision points for alternative paths. This can be used for visually describing sequential algorithms, or for simplifying otherwise complicated transitions.

- **History junctions** are used for tracking the activity of states. If a superstate containing a history junction has been activated before, instead of executing its default transition path upon entry, its last activated sub-state is entered.
- **Transition segments** are connections between charts, states and junctions. A transition segment has a source and destination, optional triggering events, an optional event guard, and optional condition actions and transition actions. A transition segment is enabled if its triggering event matches the current triggering event of the chart (or it has no triggering events), and its guard evaluates to true (or it has no guard).
- **Transition paths** are sequences of transition segments that connect charts and states. A transition path is valid, if all its transition segments are enabled.
- **Flow charts** are directed subgraphs composed of transition segments, such that – except for the source and sink nodes – all its nodes are junctions. A flow chart defines a set of transition paths between its source and sink nodes.
- **Transition segment parent** is the container state (or chart or graphical function) that contains inclusively (that is, a state contains itself) the source and the destination of the transition segment.
- **Transition path parent** is the lowest common ancestor state (or chart or graphical function) that contains inclusively the parents of each transition segment in the transition path.

Stateflow distinguishes the following types of transitions:

- **Default transitions** are transition segments that are executed upon the entry of an inactive state (or chart). The source of a default transition is the same as its parent.
- **Outer transitions** are transition segments that are executed upon the execution of an active state. The source of an outer transition is the state from which the transition points away.

- **Inner transitions** are transition segments that are executed upon the execution of an active state, if its outer transitions fail. The source of an inner transition is the same as its parent.

- **Outgoing transitions** are transitions from junctions. They are executed upon the evaluation of a flow chart.

When triggered, a Stateflow chart is executed by evaluating its active states and finding a valid transition path. This includes executing the during actions of states, and executing the condition actions of transition segments. Upon finding a valid transition path, the source state(s) are exited and the destination state(s) are entered by executing the entry and exit actions of the corresponding states, and executing the transition actions of the transition path.

Being one of the most important steps in the execution, we describe the search for a valid transition path in the following. For other steps, we refer the reader to the Stateflow user guide [77]. Given a flow chart, the search for a valid transition path is as follows:

1. Start with the source node of the flow chart,
2. evaluate the first transition segment of the node,
3. if the transition segment is enabled:
    (a) perform its condition actions,
    (b) if this inactivates the source of the flow chart, perform early return (the flow chart fails to execute),
    (c) evaluate the transition segment's destination:
        i. if the destination is a state, a valid transition path is found. Evaluate the list of transition actions along the transition path and return with success. If any of the transition actions inactivates the source of the flow chart, do not execute the rest of the actions (early return, and the flow chart fails to execute).

ii. if the destination is a terminal junction (junction without outgoing transition segments), return with success.

iii. otherwise, repeat from step 2 for the destination. If it returns with fail, continue with step 4.

4. Try the next transition segment from step 3, until there are more transition segments.

5. Otherwise, return with failure.

Upon the successful execution of a flow chart, we need to exit and enter some of the states according to the rules of Stateflow. For the exact details, see the Stateflow user guide [77]. We formalized these rules using the ForSpec language, and in the next section we will discuss excerpts from the specification.

Action language

The action language of Stateflow is an imperative language (with side-effects) similar to the C language. It is beyond the scope to discuss the complete language here, but we describe the process how variable (data) names are resolved in the language.

In Stateflow, actions are performed in a context. A context is a state, chart or graphical function: for state actions, the state itself is the execution context; for transition actions, the parent state is the context. An identifier is resolved by looking up if it is defined in the current context. If it is, the data belonging to the identifier is used. Otherwise, the identifier is recursively looked up in the ancestors of the context until it is not found.
The action language also defines additional constructs for interacting with Stateflow diagrams:

- **Send broadcast event:** broadcasts an event to the chart. This results in an embedded execution of the chart (self-recursion), after which it returns to the original execution.

- **Send local broadcast event:** broadcasts an event to a state specified as a path (list of identifiers) in the chart. This results in an embedded execution of the state.

These constructs introduce much of the challenges in the semantics of Stateflow: since they allow a chart to recursively trigger itself leading to complicated executions.

7.3.5 Structural semantics

In this section, we introduce an abstract syntax for the Stateflow language, and describe its well-formedness rules.

Syntax of Stateflow

A Stateflow model is a hierarchical state diagram, in which the root elements are *charts*. A chart is a uniquely named container that contains states, junctions and graphical functions, as well as a set of default transitions. Furthermore, a chart has a set of (local, input and output) data, and settings related to its execution semantics (single-step vs. super-step semantics and its related settings):

```
chart ::= new (name: String).
chart_type ::= fun (chart ⇒ {OR,AND}).
chart_substate ::= new (chart, state).
chart_default_transition ::= new (chart, transition).
chart_junction ::= new (chart, junction).
chart_function ::= new (chart, function).
chart_data ::= new (chart, data).
chart_event ::= new (chart, event).
chart_enableNonTerminalStates      ::= fun (chart ⇒ Boolean).
chart_nonTerminalMaxCounts         ::= fun (chart ⇒ Integer).
chart_nonTerminalUnstableBehavior ::= fun (chart ⇒ {PROCEED,
    THROW_ERROR}).
```

In Stateflow, a state is a container with a name possibly containing other substates. There are two types of states: OR and AND state. An OR state represents exclusivity:

at any moment, at most one of its substates is active. In contrast, an AND state represents concurrency: at any moment, either none, or all of its substates are active. States have default, inner and outer transitions, as well as entry, during, exit and on actions that describe the actions taken on the activation, execution, deactivation of a state, and the receipt of an event. A state is uniquely identified by its parent and its name:

```
state ::= new (parent: any state+chart, name: String).
state_type ::= fun (state ⇒ {OR,AND}).
state_order ::= fun (state ⇒ Integer).
state_default_transition ::= new (state, transition).
state_inner_transition ::= new (state, transition).
state_outer_transition ::= new (state, transition).
state_substate ::= new (state, state).
state_junction ::= new (state, junction).
state_function ::= new (state, function).
entry_actions ::= fun (state ⇒ action_list).
during_and_on_actions ::= fun (state ⇒ action_list).
exit_actions ::= fun (state ⇒ action_list).
state_data ::= new (state, data).
state_event ::= new (state, event).
```

Besides regular states, Stateflow contains connective and history junctions that are special unnamed states only used for intermittent steps. A junction is identified by its parent and its identifier. A junction has a type and a set of outgoing transitions:

```
junction ::= new (parent: any state+chart, id:String).
junction_type ::= fun (junction ⇒ {CONNECTIVE, HISTORY}).
junction_outgoing_transition ::= new (junction, transition).
```

Graphical functions are named functions that perform actions by the execution of a flowchart (a chart containing only junctions). A graphical function has a set of (local, input, and output) data:

```
function ::= new (parent:any state+chart, name:String).
function_default_transition ::= new (function, transition).
function_junction ::= new (function, junction).
function_data ::= new (function, data).
```

A transition is a directed connection between two locations. A transition has a (possible empty) list of trigger events and an optional guard condition, which together determines whether the transition is enabled at a given time instant. A transition has two types of optional actions: condition action and transition action that are executed upon the execution of the transition. The difference is that while the condition action is immediately executed upon the firing of a transition, the transition actions are collected until a valid transition path is found. The execution of a valid transition path includes the execution of the collected transition actions. Otherwise, if no valid transition path is found, the transition actions are not executed at all.

```
transition ::= new (parent: state+chart+function,
          src: chart+state+junction,
          dst: state+junction,
          ev : event_list).
transition_type ::= fun (transition ⇒
   {DEFAULT,INNER,OUTER,OUTGOING}).
transition_order ::= fun (transition ⇒ Integer).
guard ::= fun (transition ⇒ condition + {null}).
condition_actions ::= fun (transition ⇒ action_list).
transition_actions ::= fun (transition ⇒ action_list).
```

Data and events are named entities with unique owners and scopes:

```
datatypes ::=
   {DOUBLE,SINGLE,INT32,INT16,INT8,UINT32,UINT16,UINT8,BOOLEAN}.
scopes    ::= {LOCAL,INPUT,OUTPUT}.
data      ::= new (owner: state+chart+function, name: String,
   datatype:datatypes, scope:scopes).
event     ::= new (owner: state+chart, name:String, scope:scopes).
```

A path is a list of strings identifying a location in the hierarchy:

```
path ::= path_cons + {null}.
path_cons ::= new (String, any path).
```

Well-formedness rules of Stateflow

In this section, we formalize the well-formedness rules of Stateflow. A Stateflow model conforms if the following holds:

```
conforms
  no invalid_data_name,
  no invalid_event_name,
  no invalid_history_junction,
  no invalid_transition_type,
  no invalid_parent,
  no invalid_owner,
  no invalid_state_order,
  no invalid_transition_order,
  no invalid_transition_cross,
  no invalid_data(_,_),
  no invalid_path,
  no invalid_broadcast,
  no invalid_transition_parent,
  no invalid_transition.
```

In the following, we discuss the definition of each of these terms. Data and event shall have unique names within their owner.

```
invalid_data_name :- A is data, B is data, A != B, A.owner =
    B.owner, A.name = B.name.
invalid_event_name :- A is event, B is event, A != B, A.owner =
    B.owner, A.name = B.name.
```

Charts, graphical functions and parallel composition states shall have no history junctions.

```
invalid_history_junction :-
  chart_junction(_,H), junction_type(H) ⇒ HISTORY;
  function_junction(_,H), junction_type(H) ⇒ HISTORY;
  state_junction(S,H), state_type(S) ⇒ AND, junction_type(H) ⇒
   HISTORY.
```

The type of a transition shall match its usage in its parent.

```
invalid_transition_type :-
```

```
  chart_default_transition(_,T),
  transition_type(T) => Type, Type != DEFAULT.
invalid_transition_type :-
  state_default_transition(_,T),
  transition_type(T) => Type, Type != DEFAULT.
invalid_transition_type :-
  function_default_transition(_,T),
  transition_type(T) => Type, Type != DEFAULT.
invalid_transition_type :-
  state_inner_transition(_,T),
  transition_type(T) => Type, Type != INNER.
invalid_transition_type :-
  state_outer_transition(_,T),
  transition_type(T) => Type, Type != OUTER.
invalid_transition_type :-
  junction_outgoing_transition(_,T),
  transition_type(T) => Type, Type != OUTGOING.
```

The parent of each state, junction and function contained within a chart or state shall be the container itself. Conversely, each state, junction and function shall have a parent state or chart that indeed contains it.

```
invalid_parent :-
  chart_substate(P,S), P != S.parent;
  chart_junction(P,S), P != S.parent;
  chart_function(P,S), P != S.parent;
  state_substate(P,S), P != S.parent;
  state_junction(P,S), P != S.parent;
  state_function(P,S), P != S.parent;
  S is state, P = S.parent, P:chart, no chart_substate(P,S);
  S is state, P = S.parent, P:state, no state_substate(P,S);
  S is junction, P = S.parent, P:chart, no chart_junction(P,S);
  S is junction, P = S.parent, P:state, no state_junction(P,S);
  S is function, P = S.parent, P:chart, no chart_function(P,S);
  S is function, P = S.parent, P:state, no state_function(P,S).
```

The owner-child relationships shall be valid for data and events.

```
invalid_owner :-
  chart_data(P,S), P != S.owner; chart_event(P,S), P != S.owner;
```

```
  state_data(P,S), P != S.owner; state_event(P,S), P != S.owner;
  function_data(P,S), P != S.owner;
  S is data, P = S.owner, P:chart, no chart_data(P,S);
  S is data, P = S.owner, P:state, no state_data(P,S);
  S is data, P = S.owner, P:function, no function_data(P,S);
  S is event, P = S.owner, P:chart, no chart_event(P,S);
  S is event, P = S.owner, P:state, no state_event(P,S).
```

States shall have a unique ordering under a parallel composition (state or chart).

```
invalid_state_order :-
  P is chart, chart_type(P,AND), chart_substate(P,S1),
   chart_substate(P,S2), S1 != S2, state_order(S1,o1),
   state_order(S2,o2), o1 = o2;
  P is state, state_type(P,AND), state_substate(P,S1),
   state_substate(P,S2), S1 != S2, state_order(S1,o1),
   state_order(S2,o2), o1 = o2.
```

Transitions from a single source shall have different orders.

```
invalid_transition_order :-
  chart_default_transition(P,T1), transition_order(T1,o1),
  chart_default_transition(P,T2), transition_order(T2,o2),
  T1 != T2, o1 = o2.
invalid_transition_order :-
  state_default_transition(P,T1), transition_order(T1,o1),
  state_default_transition(P,T2), transition_order(T2,o2),
  T1 != T2, o1 = o2.
invalid_transition_order :-
  function_default_transition(P,T1), transition_order(T1,o1),
  function_default_transition(P,T2), transition_order(T2,o2),
  T1 != T2, o1 = o2.
invalid_transition_order :-
  state_outer_transition(P,T1), transition_order(T1,o1),
  state_outer_transition(P,T2), transition_order(T2,o2),
  T1 != T2, o1 = o2.
invalid_transition_order :-
  state_inner_transition(P,T1), transition_order(T1,o1),
  state_inner_transition(P,T2), transition_order(T2,o2),
  T1 != T2, o1 = o2.
```

```
invalid_transition_order :-
  junction_outgoing_transition(P,T1), transition_order(T1,o1),
  junction_outgoing_transition(P,T2), transition_order(T2,o2),
  T1 != T2, o1 = o2.
```

A transition shall not cross the border of an AND composition:

```
invalid_transition_cross :-
  T is transition, ancestor_or_self(T.src,A),
   ancestor(A,T.parent), state_type(A) ⇒ AND;
  T is transition, ancestor_or_self(T.dst,A),
   ancestor(A,T.parent), state_type(A) ⇒ AND.
```

All the data identifiers shall be well-defined.

```
invalid_data ::= (identifier, state+junction+function+chart).
invalid_data(Var,Context) :- actionsInContext(Actions,Context),
   subexpr(Actions,Var), undefined_var_by_name(Var,Context).
```

Each path shall refer to valid elements in the chart.

```
invalid_path :-
  guard(Trans,G), subexpr(G,IN(Path)), Path:path, no
   StateAtPath(Trans.parent,Path,_);
  actionsInContext(A,Context), subexpr(A,identifier(Path)),
   Path:path, no DataAtPath(Context,Path,_);
  actionsInContext(A,Context),
   subexpr(A,directedEventBroadcast(_,Path)), Path:path, no
   StateAtPath(Context,Path,_).
```

Only local events shall be broadcast.

```
invalid_broadcast :-
  actionsInContext(Actions,_),
   subexpr(Actions,eventBroadcast(E)), E.scope != LOCAL;
  actionsInContext(Actions,_),
   subexpr(Actions,directedEventBroadcast(E,_)), E.scope != LOCAL.
```

The parent of an inner transition or default transition shall be its source.

```
invalid_transition_parent :-
  T is transition, transition_type(T,INNER), T.src != T.parent;
```

```
  T is transition, transition_type(T,DEFAULT), T.src != T.parent.
```

The parent of a transition shall be ancestor for both the source and destination.

```
invalid_transition_parent :-
  T is transition, no ancestor_or_self(T.src,T.parent);
  T is transition, no ancestor_or_self(T.dst,T.parent).
```

A transition shall connect elements within the same chart.

```
invalid_transition :-
  T is transition,
  ancestor(T.src,Chart1), Chart1:chart,
  ancestor(T.dst,Chart2), Chart2:chart,
  Chart1 != Chart2.
```

7.3.6 Operational semantics

In this section, we discuss excerpt from the Structural Operational Semantics (SOS) of Stateflow. For brevity, we only present the formal semantics of chart execution and transition execution.

Execution Environment

First, we define an environment that stores the actual status of the system, that is,

- the set of active states,
- the set of variable valuations,
- the current triggering event and
- the last active substate for each state that has been activated before (in order to support history junctions).

We can formalize the environment with the following quadruple:

```
environment ::= (active_states: in.state_list,
          data_values: in.valuation_list,
          current_event: in.event+{null},
          last_active_substates: in.state_pair_list).
```

Chart execution

A Stateflow chart is a reactive system that produces responses to external events. In the following, we formalize the execution of a chart as specified in the Simulink user guide. The execution of an inactive chart (a chart that has only inactive states) results in the initialization of the chart, that is, the entry of the root state of the chart. After executing the inactive chart, it becomes a sleeping but active chart (unless the chart has no states, in which case every triggering results in initialization). A sleeping chart is activated by another event, in response to which the chart executes its active states. In its default single-step operation mode, the chart executes only once for each received event.

The execute_chart rule formalizes these rules. If the chart has any active substates, we execute the active chart; otherwise, we initialize the chart. The arguments for execute_chart are the chart to execute and the current environment, and it returns the updated environment after executing the chart exactly once:

```
execute_chart ::= [ in.chart, environment ⇒ environment].
```

If the chart has any active states, execute it as an active chart:

```
execute_chart(C,Env) ⇒ Env' :-
  anyActiveSubstate(C,Env) ⇒ TRUE,
  execute_active_chart(C,Env) ⇒ Env'.
```

Otherwise, activate an OR chart by evaluating its default flow path:

```
execute_chart(C,Env) ⇒ Env'' :-
  in.chart_type(C) ⇒ OR, anyActiveSubstate(C,Env) ⇒ FALSE,
  execute_default_transitions(C,Env,null) ⇒ (Env',Res),
  enter_sole_child_if_any(C,Env') ⇒ Env''.
```

And activate an AND chart by entering all its sub-states:

```
execute_chart(C,Env) ⇒ Env' :-
  in.chart_type(C) ⇒ AND, anyActiveSubstate(C,Env) ⇒ FALSE,
  in.ordered_list_of_substates(C) ⇒ List,
  enter_each_state(List,Env) ⇒ Env'.
```

The execute_active_chart rule formalizes the execution of an active chart. The arguments for execute_active_chart are the chart to execute and the current environment, and it returns the updated environment after executing the chart exactly once:

```
execute_active_chart ::= [ in.chart, environment ⇒ environment].
```

Upon the execution of an active chart, if the chart has no active sub-states, return the intact environment:

```
execute_active_chart(C,Env) ⇒ Env :-
  anyActiveSubstate(C,Env) ⇒ FALSE.
```

If a sub-state of an OR chart is active, execute that sub-state:

```
execute_active_chart(C,Env) ⇒ Env' :-
  in.chart_type(C) ⇒ OR,
  in.chart_substate(C, Child),
  is_active(Env,Child) ⇒ TRUE,
  execute_state(Child,Env) ⇒ Env'.
```

If any sub-state of an AND chart is active, execute all its sub-states:

```
execute_active_chart(C,Env) ⇒ Env' :-
  in.chart_type(C) ⇒ AND, anyActiveSubstate(C,Env) ⇒ TRUE,
  execute_each_active_substate(C,Env) ⇒ Env'.
```

The execute_each_active_substate rule describes the execution of all the substates of a state:

```
execute_each_active_substate ::= [in.chart+in.state, environment
    ⇒ environment].
// Retrieve the list of substates, and call execute_each_active_state.
execute_each_active_substate(C,Env) ⇒ Env' :-
  in.ordered_list_of_substates(C) ⇒ List,
```

```
  execute_each_active_state(List,Env) ⇒ Env'.
```

The execute_each_active_state rule describes the execution of all the states in a list:

```
execute_each_active_state ::= [in.state_list, environment ⇒
    environment].
// If the list is empty, return the intact environment.
execute_each_active_state(null,Env) ⇒ Env.
// Otherwise, execute first state in list, and repeat for the rest of the list.
execute_each_active_state(in.state_cons(S,Rest),Env) ⇒ Env'' :-
  execute_state(S,Env) ⇒ Env',
  execute_each_active_state(Rest,Env') ⇒ Env''.
```

The enter_each_state rule enters each state in a list of states:

```
enter_each_state ::= [in.state_list, environment ⇒ environment].
// If the list is empty, return the intact environment.
enter_each_state(null,Env) ⇒ Env.
// Otherwise, enter first state in list, and repeat for the rest of the list.
enter_each_state(in.state_cons(Child,Rest),Env) ⇒ Env'' :-
    enter_state(Child,null,Env) ⇒ Env',
    enter_each_state(Rest,Env') ⇒ Env''.
```

This concludes the execution of a chart.

Transition rules

The transition rules describe the execution of a flow chart in Stateflow. A transition segment interconnects two locations, and a flow chart is a directed graph formed by a set of transition segments. Stateflow distinguishes three types of flow charts:

- default flow charts that start with a default transition segment of the state;
- outer flow charts that leave the state;
- inner flow charts that stay within the state.

A transition path is a sequence of transition segments that connects two states. A valid transition path is composed of transition segments that are all enabled: the

current event triggers them, and their guards evaluate to true. A flow chart is executed by probing the transitions from the current state one-by-one (and backtracking if necessary), until either a valid transition path is found, or the list of transitions is exhausted.

We describe the semantics of a transition path by maintaining a list of constituent transition segments. If the transition path is invalid, the list is thrown away. Otherwise, at the execution of the valid transition path, they describe the states that must be entered and exited, as well as the actions to be taken.

The following rules describe the execution of default, outer and inner transitions by retrieving the corresponding list of transition segments and handing it over to the execute_transitions rule:

```
execute_default_transitions(C,Env,TL) ⇒ (Env',Res) :-
  in.list_of_default_transitions(C) ⇒ List,
  execute_transitions(List,Env,TL)  ⇒ (Env',Res).
execute_outer_transitions(C,Env,TL) ⇒ (Env',Res) :-
  in.list_of_outer_transitions(C)  ⇒ List,
  execute_transitions(List,Env,TL) ⇒ (Env',Res).
execute_inner_transitions(C,Env,TL) ⇒ (Env',Res) :-
  in.list_of_inner_transitions(C)  ⇒ List,
  execute_transitions(List,Env,TL) ⇒ (Env',Res).
```

The execute_transitions rule defines the execution of a flow chart given its initial transition segments. The arguments to the execute_transitions rule are

- a list of transitions to try,
- the current environment,
- the list of transitions that have been stored so far in the transition path.

The rule returns the updated environment after executing the flow chart, and a status flag indicating whether or not the transition successfully reached a terminal state:

```
execute_transitions ::= [in.transition_list, environment,
   in.transition_list ⇒ environment, result].
```

If there are no more transitions to try, return failed:

```
execute_transitions(null,Env,TL) ⇒ (Env,FAILED).
```

If a successful transition path was found, return the modified environment:

```
execute_transitions(in.transition_cons(T,R),Env,TL)⇒(Env',Res) :-
  execute_transition(T,Env,in.transition_cons(T,TL))⇒(Env',Res),
  Res != FAILED.
```

If the current transition failed, try the rest of the transitions:

```
execute_transitions(in.transition_cons(T,R),Env,TL)⇒(E'',Res') :-
  execute_transition(T,Env,in.transition_cons(T,TL))⇒(E',FAILED),
  execute_transitions(R,E',TL) ⇒ (E'',Res').
```

A transition segment can fire if the current event is its triggering event and its guard evaluates to true. On firing, the condition action is immediately executed, and the transition action is stored in a list of actions, which are executed if a successful transition path is found. The context for the condition action is the parent of the transition segment. Normally, after the condition action, the destination of the transition is executed. Otherwise, if the condition action inactivates the source state of the transition path under examination, the destination is not executed, and the environment produced by the condition action along with the status flag ‘FAILED‘ is returned. If a node cannot fire, it returns the intact environment along with the status flag ‘FAILED‘.

The arguments to the transition operation are

- the transition segment,
- the current environment,
- the list of transitions that have been stored so far in the transition path.

The result of a transition segment is a pair that describes the updated environment and a status flag indicating whether or not the transition successfully reached a terminal state:

```
execute_transition  ::= [ in.transition, environment,
    in.transition_list ⇒ environment, result].
```

Executing an enabled transition segment:

```
execute_transition(T,Q,TL) ⇒ (Q'',Result) :-
  is_any_event_triggered(Q,T.ev) ⇒ TRUE,      // a transition event is triggered
  evaluate_guard(T,Q) ⇒ TRUE,                 // guard evaluates to true
  condition_actions(T,Q) ⇒ Q',                // executing condition action
  destination(T.dst,Q',TL) ⇒ (Q'', Result).  // execute the destination
```

If either the transition event is not triggered, or the guard evaluates to false, the transition is disabled:

```
execute_transition(T,Q,TL) ⇒ (Q,FAILED) :-
  is_any_event_triggered(Q,T.ev) ⇒ FALSE.
execute_transition(T,Q,TL) ⇒ (Q,FAILED) :-
  is_any_event_triggered(Q,T.ev) ⇒ TRUE,
  evaluate_guard(T,Q) ⇒ FALSE.
```

This concludes the execution of a transition segment, a transition path and a flow chart.

7.3.7 Testing

In order to test the specifications, we developed a testing harness in Matlab. The workflow of the testing is shown in Fig. 11. The testing environment consists of a set of unit test models and scripts for translating the tests to ForSpec and executing them. We used the Matlab Stateflow user guide to extract test cases that provide a good coverage of the language: most of them exercise corner cases of the language. We also designed several other test cases, such as the example shown at the beginning of this section. Overall, we developed 20 test cases, all of which were passed by the specifications.

In order to be able to compare the output produced by Matlab versus the output produced by our semantic specifications, we wrote a parser using the Matlab script

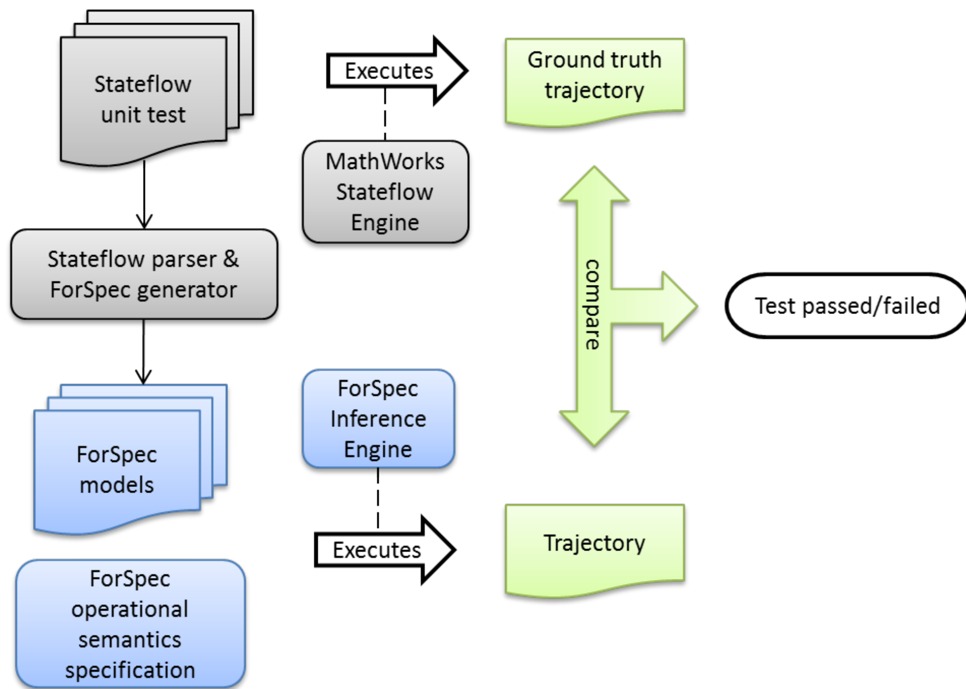


Figure 11: Testing harness for the specifications.

language that can translate Matlab Stateflow models to ForSpec models. After performing the translation, we can execute the tests by ForSpec and compare it to the execution by Matlab Stateflow.

Such a testing harness greatly helped the development of the semantic specifications: after changing the specifications we could immediately run the tests to see if the modification has broken any of them.

7.3.8 Conclusion

In this section, we demonstrated the formal semantic specification of the Matlab Stateflow language. The complete specification with the English language documentation is approximately 2000 lines long. This includes both the diagram language and the action language, their abstract syntaxes, and their structural and behavioral semantic specifications.

The sections discussing the structural and operational semantics were directly generated from the specifications, which shows the advantage of writing specifications in a literate programming style: we can easily keep the documentation up-to-date. In particular, the same specification is used by the testing harness and the generated documentation.

CHAPTER 8

CONCLUSION

In this thesis, we discussed the formal semantic specification of CPS DSMLs. The motivation for developing formal semantics for languages is to provide unambiguous documentation, and to facilitate the interoperability of the tools of the languages.

On one hand, for safety-critical CPS, we need formal guarantees that the developed systems are safe, therefore we need tool support for verifying properties of the CPS models. On the other hand, we wish to leverage the same models for automatically generating controller code, performing design-space exploration and other functions. By developing formal semantic specifications for the languages used by the models, we can develop tools that are based on common interpretations - the unambiguous semantics of the languages. It remains a future work to use these specifications for the above-mentioned functions.

Our contribution is the following:

- ForSpec, a specification language with support for operational, denotational and translational style specifications.
- A set of reusable semantic units for CPS languages that facilitate the denotational semantic specification of acausal physical modeling languages, controller design languages and hybrid languages.
- Identification of different specification styles in ForSpec and FORMULA, such as denotational, operational and translational styles.
- Three case studies discussing the specification of a physical modeling language, a CPS integration language and a controller design language.

- Tool support for executing the ForSpec specifications, translating them to FORMULA specifications, and for generating documentation from the specifications.

There are several directions for future work. The most plausible one is the usage of the FORMULA verification tool to perform bounded model checking for actual controller models. Based on our previous experiences, the Satisfiability Modulo Theories (SMT) encoding of the problem greatly influences the scalability of the approach. We expect that by further advancement of the FORMULA tool, we may reach a point, where the tool can be efficiently used for performing the verification of the models based on their formal semantic specifications.

Another important future work is the semantic specification of concurrent processes. Since FORMULA performs forward-inference until reaching a fix-point, the execution of a concurrent specification leads to the simultaneous enumeration of all the possible trajectories. It needs further work to evaluate the scalability of this approach.

In conclusion, we provided a discussion of the formal semantic specification of CPS modeling languages that serves as a mathematically rigorous foundation for these languages and their tools. Providing this foundation is an important step towards the design and implementation of provably safe Cyber-Physical Systems.

REFERENCES

[1] T. Abdellatif, J. Combaz, and J. Sifakis. “Model-based implementation of real-time applications”. In: *Proceedings of the tenth ACM international conference on Embedded software*. 2010, 229–238.

[2] G. A. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1985.

[3] A. Agrawal, G. Simon, and G. Karsai. “Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations”. In: *Electronic Notes in Theoretical Computer Science* 109 (2004), pp. 43–56.

[4] R. Alur and D. L. Dill. “A theory of timed automata”. In: *Theoretical computer science* 126.2 (1994), 183–235.

[5] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. “Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems”. In: *Hybrid Systems*. Vol. 736. Springer Berlin Heidelberg, 1993, pp. 209–229.

[6] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. “UML2Alloy: A Challenging Model Transformation”. In: *Model Driven Engineering Languages and Systems*. Vol. 4735. Springer Berlin Heidelberg, 2007, pp. 436–450.

[7] M. Anlauff. “X asm-an extensible, component-based abstract state machines language”. In: *Abstract State Machines-Theory and Applications*. Springer. 2000, pp. 69–90.

[8] M. Association. *Modelica - A Unified Object Oriented Language for Physical System Modeling, Language Specification, Version 3.3*. https://www.modelica.org/documents. Accessed 12 February 2014.

[9] J. W. Backus. “The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference”. In: *Proceedings of the International Conference on Information Processing, 1959* (1959).

[10] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. “The graph rewriting and transformation language: GReAT”. In: *Electronic Communications of the EASST* 1 (2007).

[11] R. Balmer. *Thermodynamics*. West Publishing, St. Paul, 1990.

[12] L. Baresi and P. Spoletini. “On the use of Alloy to analyze graph transformation systems”. In: *Graph Transformations*. Springer, 2006, 306–320.

[13] A. Basu, M. Bozga, and J. Sifakis. “Modeling Heterogeneous Real-time Components in BIP”. In: *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*. 2006, pp. 3–12.

[14] M. Berz and K. Makino. “Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models”. In: *Reliable Computing* 4.4 (1998), pp. 361–369.

[15] S. Bliudze and J. Sifakis. “The Algebra of Connectors – Structuring Interaction in BIP”. In: *IEEE Transactions on Computers* 57.10 (Oct. 2008), pp. 1315 –1330.

[16] E. Börger and R. F. Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Verlag, 2003.

[17] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. “A high-level modular definition of the semantics of C?” In: *Theoretical Computer Science* 336.2 (2005), pp. 235–284.

[18] E. Börger and D. Rosenzweig. “A mathematical definition of full Prolog”. In: *Science of Computer Programming* 24.3 (1995), pp. 249–286.

[19] A. Boronat and J. Meseguer. “An algebraic semantics for MOF”. In: *Fundamental Approaches to Software Engineering* (2008), 377–391.

[20] B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France, and G. Karsai. “Challenges and directions in formalizing the semantics of modeling languages”. In: *Computer Science and Information Systems/ComSIS* 8.2 (2011), pp. 225–253.

[21] K. Chen, J. Sztipanovits, and S. Neema. “Compositional specification of behavioral semantics”. In: *Proceedings of the conference on Design, automation and test in Europe*. DATE ’07. EDA Consortium, 2007, 906–911.

[22] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. “Semantic Anchoring with Model Transformations”. In: *Model Driven Architecture – Foundations and Applications*. Vol. 3748. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, pp. 115–129.

[23] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[24] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. “Maude: Specification and programming in rewriting logic”. In: *Theoretical Computer Science* 285.2 (2002), 187–243.

[25] Z. Demirezen, M. Mernik, J. Gray, and B. Bryant. “Verification of DSMLs using graph transformation: a case study with Alloy”. In: *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*. 2009, p. 3.

[26] L. De Moura and N. Bjørner. “Z3: An efficient SMT solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[27] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. “Extending AMMA for supporting dynamic semantics specifications of DSLs”. In: (2006).

[28] *EcosimPro*. `www.ecosimpro.com`. Accessed 12 February 2014.

[29] M. Egea and V. Rusu. “Formal executable semantics for conformance in the MDE framework”. In: *Innovations in Systems and Software Engineering* 6.1-2 (2009), pp. 73–81.

[30] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1: Equations and initial semantics*. Springer-verlag, 2008.

[31] N. Esfahani, S. Malek, J. P. Sousa, H. Gomaa, and D. A. Menascé. “A modeling language for activity-oriented composition of service-oriented software systems”. In: *Model Driven Engineering Languages and Systems*. Springer, 2009, 591–605.

[32] A. Evans, R. France, K. Lano, and B. Rumpe. “The UML as a formal modeling notation”. In: *The Unified Modeling Language. UML’98: Beyond the Notation*. Springer, 1999, 336–348.

[33] A. Evans, R. France, and E. Grant. “Towards formal reasoning with uml models”. In: *Proceedings of the OOPSLA*. Vol. 99. 1999.

[34] R. W. Floyd. “Assigning meanings to programs”. In: *Mathematical aspects of computer science* 19.19-32 (1967), p. 1.

[35] P. Fritzson and V. Engelson. “Modelica — A unified object-oriented language for system modeling and simulation”. In: *ECOOP’98 — Object-Oriented Programming*. Vol. 1445. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1998, pp. 67–90.

[36] A. Gargantini, E. Riccobene, and P. Scandurra. “A semantic framework for metamodel-based languages”. In: *Automated Software Engineering* 16.3-4 (Apr. 2009), pp. 415–454.

[37] A. Goderis, C. Brooks, I. Altintas, E. Lee, and C. Goble. “Composing Different Models of Computation in Kepler and Ptolemy II”. In: *Computational Science – ICCS 2007*. Vol. 4489. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, pp. 182–190.

[38] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. “Heterogeneous composition of models of computation”. In: *Future Generation Computer Systems* 25.5 (May 2009), pp. 552–560.

[39] C. A. Gunter, P. D. Mosses, and D. S. Scott. *Semantic domains and denotational semantics*. Tech. rep. DTIC Document, 1989.

[40] Y. Gurevich. “Evolving algebras 1993: Lipari guide”. In: *Specification and validation methods* (1995), 9–36.

[41] Y. Gurevich, B. Rossman, and W. Schulte. “Semantic essence of AsmL”. In: *Theoretical Computer Science* 343.3 (Oct. 2005), pp. 370–412.

[42] Y. Gurevich and J. K. Huggins. “The semantics of the C programming language”. In: *Computer Science Logic*. Springer. 1993, pp. 274–308.

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. “The synchronous data flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[44] G. Hamon. “A denotational semantics for stateflow”. In: *Proceedings of the 5th ACM international conference on Embedded software*. EMSOFT ’05. ACM, 2005, 164–172.

[45] G. Hamon and J. Rushby. “An Operational Semantics for Stateflow”. In: *Fundamental Approaches to Software Engineering*. Vol. 2984. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, pp. 229–243.

[46] D. Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of computer programming* 8.3 (1987), 231–274.

[47] K. Hayhurst, D. Veerhusen, J. Chilenski, and L. Rierson. *A practical tutorial on modified condition/decision coverage*. Tech. rep. TM-2001-210876. NASA Langley Research Center, 2001.

[48] P. R. Henriques, M. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. “Automatic generation of language-based tools using the LISA system”. In: *Software, IEE Proceedings-*. Vol. 152. 2. IET. 2005, pp. 54–69.

[49] T. Henzinger. “The Theory of Hybrid Automata”. In: *Verification of Digital and Hybrid Systems*. Ed. by M. Inan and R. Kurshan. Vol. 170. NATO ASI Series. Springer Berlin Heidelberg, 2000, pp. 265–292.

[50] T. Henzinger, B. Horowitz, and C. Kirsch. “Giotto: A Time-Triggered Language for Embedded Programming”. In: *Embedded Software*. Ed. by T. Henzinger and C. Kirsch. Vol. 2211. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, pp. 166–184.

[51] C. A. R. Hoare. “An axiomatic basis for computer programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[52] C. A. R. Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 21.8 (1978), 666–677.

[53] D. Jackson. *Software Abstractions: Logic, Language, And Analysis*. en. MIT Press, 2006.

[54] E. K. Jackson. “The Structural Semantics of Model-based Design: Theory and Applications”. PhD thesis. Vanderbilt University, 2007.

[55] E. K. Jackson, N. Bjørner, and W. Schulte. “Canonical regular types”. In: *ICLP (Technical Communications)* (2011), 73–83.

[56] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. “Components, platforms and possibilities: towards generic automation for MDA”. In: *Proceedings of the tenth ACM international conference on Embedded software*. EMSOFT ’10. ACM, 2010, 39–48.

[57] E. Jackson and J. Sztipanovits. “Formalizing the structural semantics of domain-specific modeling languages”. In: *Software and Systems Modeling* 8.4 (2009), pp. 451–478.

[58] E. Jackson, N. Bjorner, and W. Schulte. *Open-World Logic Programs: A New Foundation for Formal Specifications*. Tech. rep. MSR-TR-2013-55. Microsoft Research, 2013.

[59] E. Jackson, T. Levendovszky, and D. Balasubramanian. “Reasoning about Metamodeling with Formal Specifications and Automatic Proofs”. In: *Model Driven Engineering Languages and Systems*. Vol. 6981. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 653–667.

[60] E. Jackson, R. Thibodeaux, J. Porter, and J. Sztipanovits. “Semantics of Domain-Specific Modeling Languages”. In: *Model-Based Design for Embedded Systems* 1 (2009), p. 437.

[61] G. Kahn. “Natural semantics”. In: *STACS 87*. Ed. by F. Brandenburg, G. Vidal-Naquet, and M. Wirsing. Vol. 247. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1987, pp. 22–39.

[62] D. Karnopp, D. L. Margolis, and R. C. Rosenberg. *System dynamics modeling, simulation, and control of mechatronic systems*. English. John Wiley & Sons, 2012.

[63] G. Karsai and J. Sztipanovits. “Model-integrated development of cyber-physical systems”. In: *Software Technologies for Embedded and Ubiquitous Systems*. Springer, 2008, 46–54.

[64] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. “On the use of graph transformation in the formal specification of model interpreters”. In: *Journal of Universal Computer Science* 9.11 (2003), 1296–1321.

[65] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. “Model-integrated development of embedded software”. In: *Proceedings of the IEEE* 91.1 (2003), 145–164.

[66] C. M. Kirsch and A. Sokolova. “The Logical Execution Time Paradigm”. In: *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[67] Z. Lattmann, A. Nagel, J. Scott, K. Smyth, J. Ceisel, C. vanBuskirk, J. Porter, T. Bapty, S. Neema, D. Mavris, and J. Sztipanovits. “Towards Automated Evaluation of Vehicle Dynamics in System-Level Designs”. In: *ASME 32nd Computers and Information in Engineering Conference (IDETC/CIE)*. 2012.

[68] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. “The generic modeling environment”. In: *Workshop on Intelligent Signal Processing, Budapest, Hungary*. Vol. 17. 2001.

[69] E. A. Lee. “Computing needs time”. In: *Communications of the ACM* 52.5 (2009), pp. 70–79.

[70] E. A. Lee. “Cyber physical systems: Design challenges”. In: *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. 2008, 363–369.

[71] E. A. Lee and H. Zheng. “Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems”. In: *Proceedings of the 7th ACM - IEEE international conference on Embedded software*. EMSOFT ’07. ACM, 2007, 114–123.

[72] S. Liang and P. Hudak. “Modular denotational semantics for compiler construction”. In: *Programming Languages and Systems (ESOP)*. Springer, 1996, pp. 219–234.

[73] N. Lynch, R. Segala, and F. Vaandrager. “Hybrid I/O automata”. In: *Information and Computation* 185.1 (Aug. 2003), pp. 105–157.

[74] O. Maler, Z. Manna, and A. Pnueli. “From Timed to Hybrid Systems”. In: *Real-Time: Theory in Practice*. Lecture Notes in Computer Science. Springer, 1992, pp. 447–484.

[75] N. Martí-Oliet and J. Meseguer. “Rewriting logic as a logical and semantic framework”. In: *Electronic Notes in Theoretical Computer Science* 4 (1996), 190–225.

[76] MathWorks. *Simscape*. `http://www.mathworks.com/products/simscape`. Accessed 12 February 2014.

[77] MathWorks. *Stateflow and Stateflow Coder User Guide (Version 5)*. `http://www.mathworks.com/help/releases/R13sp2/pdf_doc/stateflow/sf_ug.pdf`. 2003.

[78] J. Meseguer. “Twenty years of rewriting logic”. In: *The Journal of Logic and Algebraic Programming* (2012).

[79] Microsoft Research. *AsmL: Abstract State Machine Language*. `http://research.microsoft.com/en-us/projects/asml`. Accessed 12 February 2014.

[80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982.

[81] E. Moggi. *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1990.

[82] P. Mosses. *Action semantics*. Vol. 26. Cambridge University Press, 2005.

[83] P. D. Mosses. "Compiler generation using denotational semantics". In: *Mathematical Foundations of Computer Science 1976*. Springer, 1976, pp. 436–441.

[84] P. D. Mosses. "Modular structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 195–228.

[85] P. D. Mosses. "The varieties of programming language semantics and their uses". In: *Perspectives of System Informatics*. Springer. 2001, pp. 165–190.

[86] P. D. Mosses. "VDM semantics of programming languages: Combinators and monads". In: *Formal methods and hybrid real-time systems*. Springer, 2007, pp. 483–503.

[87] P. J. Mosterman and G. Biswas. "A theory of discontinuities in physical system models". In: *Journal of the Franklin Institute* 335.3 (Apr. 1998), pp. 401–439.

[88] P. Mosterman, G. Simko, and J. Zander. "A Hyperdense Semantic Domain for Discontinuous Behavior in Physical System Modeling". In: *Compositional Multi-Paradigm Models for Software Development*. Oct. 2013.

[89] P. Mosterman, G. Simko, J. Zander, and Z. Han. "A Hyperdense Semantic Domain for Hybrid Dynamic Systems to Model With Impact." In: *17th International Conference on Hybrid Systems: Computation and Control (HSCC)*. 2014.

[90] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. "The simulation semantics of SystemC". In: *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*. IEEE. 2001, pp. 64–70.

[91] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. "Weaving executability into object-oriented meta-languages". In: *Model Driven Engineering Languages and Systems*. Springer, 2005, pp. 264–278.

[92] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/-Transformation (QVT)*. `http://www.omg.org/spec/QVT/Current`. Accessed 12 February 2014.

[93] S. Owre, J. M. Rushby, and N. Shankar. "PVS: A prototype verification system". In: *Automated Deduction—CADE-11*. Springer, 1992, 748–752.

[94] S. Owre, N. Shankar, and J. M. Rushby. "The PVS specification language (beta release)". In: *Computer Science Laboratory, SRI International, Menlo Park, CA* 1020 (1993).

[95] R. F. Paige, P. J. Brooke, and J. S. Ostroff. "Metamodel-based model conformance and multiview consistency checking". In: *ACM Transactions on Software Engineering and Methodology* 16.3 (July 2007), 11–es.

[96] C. A. Petri and W. Reisig. "Petri net". In: *Scholarpedia*. Vol. 3. 4. 2008, p. 6477.

[97] G. D. Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[98] G. D. Plotkin. "A structural approach to operational semantics". In: *The Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 17–140.

[99] G. D. Plotkin. "The origins of structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60 (2004), pp. 3–15.

[100] J. Porter, G. Hemingway, Nine, H., van Buskirk, C., Kottenstette, N., Karsai, G., and Sztipanovits, J. *The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design. Part 1: Design Language, Modeling Framework, and Analysis*. Tech. rep. ISIS-10-109. ISIS, 2010.

[101] M. R. Raskovsky. "Denotational semantics as a specification of code generators". In: *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. Vol. 17. 6. ACM, 1982.

[102] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. "A definition and classification of timing anomalies". In: *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*. 2006.

[103] J. E. Rivera and A. Vallecillo. "Adding Behavior to Models". In: IEEE, Oct. 2007, p. 169.

[104] J. E. Rivera, F. Durán, and A. Vallecillo. "Formal Specification and Analysis of Domain Specific Models Using Maude". en. In: *SIMULATION* 85.11-12 (Nov. 2009), pp. 778–792.

[105] J. E. Rivera, F. Durán, and A. Vallecillo. "On the behavioral semantics of real-time domain specific visual languages". In: *Rewriting Logic and Its Applications*. Springer, 2010, pp. 174–190.

[106] J. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. "Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude". In: *Software Language Engineering*. Vol. 5452. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 54–73.

[107] J. R. Romero, J. E. Rivera, F. Durán, and A. Vallecillo. "Formal and Tool Support for Model Driven Engineering with Maude." In: *The Journal of Object Technology* 6.9 (2007), pp. 187–207.

[108] G. Rosu. *Programming Languages – A Rewriting Approach (draft)*. `http://fsl.cs.uiuc.edu/pub/pl.pdf`. Accessed 12 February 2014.

[109] G. Rosu and T. F. Serbanuta. "K Overview and SIMPLE Case Study". In: *Proceedings of International K Workshop (K'11)*. ENTCS. To appear. Elsevier, 2013.

[110] J. Rushby. "Automated test generation and verified software". In: *Verified Software: Theories, Tools, Experiments*. Springer, 2008, pp. 161–172.

[111] V. Rusu. "Embedding domain-specific modelling languages in maude specifications". In: *ACM SIGSOFT Software Engineering Notes* 36.1 (2011), 1–8.

[112] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. "Defining and translating a safe subset of simulink/stateflow into lustre". In: *Proceedings of the 4th ACM international conference on Embedded software*. ACM. 2004, pp. 259–268.

[113] D. Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group, 1970.

[114] D. Scott. "The lattice of flow diagrams". In: *Symposium on semantics of algorithmic languages*. Springer. 1971, pp. 311–366.

[115] D. Scott and C. Strachey. *Toward a mathematical semantics for computer languages*. Vol. 1. Oxford University Computing Laboratory, Programming Research Group, 1971.

[116] G. Simko, D. Lindecker, T. Levendovszky, E. K. Jackson, S. Neema, and J. Sztipanovits. “A Framework for Unambiguous and Extensible Specification of DSMLs for Cyber-Physical Systems”. In: *IEEE 20th International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*. 2013.

[117] G. Simko, T. Levendovszky, S. Neema, E. Jackson, T. Bapty, J. Porter, and J. Sztipanovits. “Foundation for Model Integration: Semantic Backplane”. In: *ASME 32nd Computers and Information in Engineering Conference (IDETC/-CIE)*. 2012.

[118] G. Simko and E. Jackson. “A Bounded Model Checking Tool for Periodic Sample-hold Systems.” In: *17th International Conference on Hybrid Systems: Computation and Control (HSCC)*. 2014.

[119] G. Simko, D. Lindecker, T. Levendovszky, S. Neema, and J. Sztipanovits. “Specification of Cyber-Physical Components with Formal Semantics – Integration and Composition”. In: *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2013.

[120] K. Slonneger and B. L. Kurtz. *Formal syntax and semantics of programming languages*. Addison-Wesley Reading, 1995.

[121] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., 1992.

[122] C. Strachey. “Towards a formal semantics”. In: *Formal Language Description Languages for Computer Programming, Proc. IFIP TC2 Conference*. 1964, pp. 198–220.

[123] C. Strachey and C. P. Wadsworth. “Continuations: A mathematical semantics for handling full jumps”. In: *Higher-order and symbolic computation* 13.1 (2000), pp. 135–152.

[124] J. Sztipanovits and G. Karsai. “Model-integrated computing”. In: *Computer* 30.4 (1997), 110–111.

[125] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and S. Wang. “Toward a Science of Cyber-Physical System Integration”. In: *Proceedings of the IEEE* 100.1 (2012), pp. 29–44.

[126] G. Taentzer. “AGG: A graph transformation environment for modeling and validation of software”. In: *Applications of Graph Transformations with Industrial Relevance*. Springer, 2004, pp. 446–453.

[127] R. D. Tennent. “The denotational semantics of programming languages”. In: *Communications of the ACM* 19.8 (1976), pp. 437–453.

[128] E. Torlak and D. Jackson. “Kodkod: A relational model finder”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 632–647.

[129] M. Utting. *Data structures for Z testing tools*. Department of Computer Science, University of Waikato, 2001.

[130] G. Wachsmuth. “Modelling the Operational Semantics of Domain-Specific Modelling Languages”. In: *Generative and Transformational Techniques in Software Engineering II*. Lecture Notes in Computer Science 5235. Springer Berlin Heidelberg, Jan. 2008, pp. 506–520.

[131] M. Whalen. “A Parametric Structural Operational Semantics for Stateflow, UML Statecharts, and Rhapsody”. In: *Month* (2010).

[132] J. Willems. “The Behavioral Approach to Open and Interconnected Systems”. In: *IEEE Control Systems* 27.6 (2007), pp. 46 –99.

[133] G. Winskel. *The formal semantics of programming languages: an introduction*. The MIT Press, 1993.

[134] R. Wrenn, A. Nagel, R. Owens, H. Neema, F. Shi, K. Smyth, D. Yao, J. Ceisel, J. Porter, C. vanBuskirk, S. Neema, T. Bapty, D. Mavris, and J. Sztipanovits. "Towards Automated Exploration and Assembly of Vehicle Design Models". In: *ASME 32nd Computers and Information in Engineering Conference (IDETC/CIE)*. 2012.