BEHAVIORAL SEMANTICS OF MODELING LANGUAGES: A PRAGMATIC APPROACH

By

Daniel Balasubramanian

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

May, 2011
Nashville, Tennessee

Approved:
Professor Gábor Karsai
Professor János Sztipánovits
Professor Douglas Schmidt
Professor Gautam Biswas
Dr. Ethan Jackson

To my family.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Gábor Karsai, for his guidance, patience and encouragement. His work ethic is a source of inspiration and his breadth of knowledge is remarkable. There was never an occasion that I went to him with a problem that he did not have the time to help me.

I would also like to thank Ethan Jackson, who has been both a friend and a colleague. His dedication to his work is amazing. He is always more than happy to discuss research at any time of day (or night!)

Part of this work was done in collaboration with NASA Ames Research Center, which included many fruitful discussions with Michael Lowry and Corina Păsăreanu, and also Michael Whalen of the University of Minnesota. Gábor Pap and Harmon Nine both made this project fun.

This work was supported by the NASA Model Transformation and Verification project and a gift from Microsoft Research.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

Chapter

I. INTRODUCTION 1

- Motivation and Challenges 3
- Contributions 5
- Outline 6

II. BACKGROUND 7

- Modeling languages 7
- Semantics 9
  - Denotational Semantics 11
  - Operational Semantics 12
  - Axiomatic Semantics 13
  - Behavioral Semantics 14
- Formal analysis methods 15
  - Abstract Interpretation 15
  - Model Checking 18
  - Theorem Proving 20
  - Test-Case Generation 22
- Statecharts 24

III. STATECHART ANALYSIS FRAMEWORK 27

- Overview 32
- Translation of structure 35
  - Data and scopes 35
  - States 38
  - Transitions 43
  - Pseudostates 43
  - Stateflow graphical functions 44
  - Data interface 47
- Execution engine 50
- Property specification 52

Scopes 55
Patterns 61
Statechart analysis with Java Pathfinder 64
Case Study 70

IV. SEMANTICS WITH FORMULA 76

Execution traces 78
Petri nets 78
Introduction to Formula 80
Behavioral semantics in Formula 84
Execution traces with Formula 88
Sequencing transformations 88
Specifying non-determinism in transformations 90
Storing traces 91
Visualizing execution traces 94

V. CONCLUSION 97

Future directions 98

Appendix

A. DIFFERENCES BETWEEN SEMANTIC VARIANTS 101

B. BEHAVIORAL SEMANTICS OF PETRI NETS IN FORMULA 102

REFERENCES 104

LIST OF TABLES

Table		Page
1	Analysis results for the MER arbiter case study.	75
2	Differences between Statechart variants.	101

LIST OF FIGURES

Figure Page

1 An example Statechart. 26
2 Example in which simple search is inefficient. 30
3 Statechart analysis framework. 33
4 Example Statechart with data used by substate. 36
5 Class diagram of main Statechart features. 37
6 Translation of a basic state. 38
7 Translation of a orthogonal states. 39
8 Translation of parallel states. 41
9 Translation of a transition. 42
10 Translation of a pseudostate. 44
11 A graphical function state named F that takes two arguments, a and b, and returns their difference. The three junctions are named j1, j2 and j3. 45
12 The generated Java code for the graphical function in Figure 11. 46
13 A Statechart and its generated data interface. 49
14 A class diagram for properties represented as scopes and patterns. 53
15 Pattern scopes. 54
16 Pattern hierarchy. 55
17 The Scope interface and base Pattern class. 56
18 The Java class for the global scope. 57
19 The Java class for the after scope. 58
20 The Java class for the before scope. 59
21 The Java class for the until scope. 60
22 The Java class for the between scope. 62
23 The Java class for the universality pattern. 63
24 The Java class for the existence pattern. 65
25 The Java class for the precedence pattern. 66
26 Sample chart for symbolic execution. 67
27 Generated structure code for the Statechart in Figure 26. 68
28 Sample output log from Symbolic Pathfinder's analysis of the model in Figure 26. 71
29 High-level view of the MER Arbiter. 72
30 Statechart for User1 from Figure 29. 73
31 A sample Petri net. Places are represented by circles, transitions by rectangles. A solid dot represents a token. Given the initial marking on the left, transition T1 firing results in the Petri net on the upper right, while the model on the bottom right is the result if transition T2 fires. 80
32 The two execution traces of the Petri net shown in Figure 31. 81
33 Formula definition of a domain for Petri nets. 83
34 Extended domain that includes Markings. 83
35 Formula encoding of the Petri net on the left side of Figure 31. 84

36 Example transformation in Formula. 87
37 A model named $m0$, which when given as input to the transformation $CreateMarkings$ in Figure 36 results in the model $m1$ in Figure 35. . . . 87
38 Trace calculation algorithm. 89
39 Algorithm to get all valid transformations for an input model. 90
40 Algorithm to get the non-deterministic choices in a model. 92
41 Algorithm to store traces. 93
42 Screenshot of how the trace tool visualizes execution trace 1 in Figure 32. 95
43 Screenshot of how the trace tool visualizes execution trace 2 in Figure 32. 96
44 The behavioral semantics for Petri nets using the MarkedPetriNet domain. 103

CHAPTER I

INTRODUCTION

A model is a formal structure that represents selected aspects of an engineering artifact and its environment. That is, models provide *abstractions* of real-world objects that allow certain details to be ignored. In doing so, models allow humans and computers to focus on the relevant features for the task at hand. For example, a model of a cruise controller for a car might capture properties such as the gear-ratios and weight, while ignoring irrelevant details, such as color and style. Using a model allows the cruise controller to be simulated, tested and verified without requiring an automobile. Physical systems, such as buildings and bridges, are modeled before they are built so that they can be rigorously analyzed to ensure their stability in the physical world. Thus, modeling can provide significant benefits in terms of time, cost and safety.

Software system modeling languages provide abstractions that are used to cope with the rising complexity of modern software. They allow designers to ignore implementation details and instead focus on the system at a high level. Software system models are used for a variety of purposes, including testing, simulation and performance analysis. They can also be refined and transformed, providing the ability to automatically synthesize certain parts of the implementation [1].

Increasingly, *domain-specific languages* (DSLs), specialized languages with concepts and features that are specific to a particular problem domain, are being used to model software systems [2]. DSLs raise the abstraction level by providing a language that is tailored to a specific area. By exposing high-level features and concepts, DSLs can express domain-specific information in a direct and compact way. When a domain-specific language is used for modeling, it is usually referred to as a *domain-specific modeling language*, or DSML. Several commercial tools are based on the concept of a DSML, including Matlab/Simulink, Modelica and LabVIEW. Each of these tools has a DSML component that allows physical systems to be modeled at a high-level using features and concepts found directly in the physical systems domain.

Custom DSMLs are used to bring the advantages of high abstraction levels to other target environments. There are a number of different tools for creating DSMLs, including the Generic Modeling Environment (GME) [3], Kermeta [4] and the Microsoft DSL Tools [5]. The creation of a DSML typically involves the definition of abstract and concrete syntax, along with structural constraints that reject erroneous instances of the DSML syntax [6]. Models conforming to a DSML can then be created.

DSMLs provide a number of benefits, but they still have issues that are largely unresolved. One of these is the difficulty of applying *formal verification* to DSMLs. Formal verification methods attempt to mechanically prove that a model's execution is correct with respect to a given specification. Examples of formal verification methods include deductive methods, model checking and static program analysis [7]. However, using any of these formal analysis methods to prove properties about a model's execution first requires a suitable method to describe the execution.

A major challenge in applying formal analysis to modeling languages is that there is no standardized way to assign behavioral semantics to a modeling language. The behavioral semantics of a language are used to describe a model's execution. Analysis tools use the description of the execution provided by the behavioral semantics to check properties. Without a precise, behavioral semantics that describes the execution of the model, one cannot perform analysis. Contrast this to traditional programming languages, such as Java, which have a well-established and standardized execution semantics [8]. Further, languages such as Java evolve very slowly, whereas modeling languages tend to be developed quickly and evolve rapidly. This rapid evolution adds an additional challenge when considering methods for assigning semantics.

Another challenge when applying formal analysis to modeling languages is the mismatch between the input languages of verification tools and modeling languages. Using an existing verification tool to analyze a modeling language requires a translation of the behavioral semantics into a form that the analysis tool understands, as well as a specification of properties to check in a language the analysis tools understands. These are both hard problems [9,10]. Several types of verification methods exist, but in practice, implementing these from scratch so that they are efficient and scalable is very difficult [11].

With these considerations and challenges in mind, the thesis of this dissertation is that modeling languages should support behavioral semantics that make them directly amenable to formal verification methods. These methods should be automated as much as possible and should allow the semantics to serve as executable specifications that clarify ambiguities found in informal documentation.

Motivation and Challenges

This work is motivated in-part by a real-world problem faced by engineers at NASA. Distributed teams of engineers there use different variants of Statecharts [12] to describe interacting pieces of a single system. Statecharts is a modeling language for describing reactive systems, which are systems that maintain an ongoing interaction with their environment. The different variants of Statecharts are similar, but have small, crucial semantic differences. These slight semantic differences across the variants mean that a model created by one team and executed with one semantics may behave very differently when executed by a different team using another semantics.

To ensure that interacting models behave as expected, a unified environment in which models of each different semantic variant can be executed and verified is needed. In considering such an environment, two relevant questions arise, both of which are directly related to the thesis statement above.

1. How can the behavioral semantics of the different variants be described precisely?

2. How can formal analysis methods be applied to verify the execution of the models?

Another motivation and separate piece of this work came from an interest in extending Formula [6,13], a modeling language and analysis tool, with the capability to compute and store execution traces of models. An execution trace is a sequence of models that shows how the state of a model evolves during the model's execution. Computing execution traces allows one to reason about the possible behaviors of a model, even when the behavior can be non-deterministic.

One challenge that had to be addressed to add this capability to Formula was creating a separate execution trace for each possible choice of applicable actions at a given step. Behavioral semantics in Formula are defined as a set of model transformations, each of which takes one model as input and produces one model as output. Each transformation represents one discrete and atomic "step" of execution. At a given point in execution, there may be multiple transformations that can be applied to an input model. For instance, in a modeling language for distributed systems, one atomic execution step might consist of adding a node to the network, while another atomic step might consist of firing a node. This can be modeled in Formula using two different transformations: one that adds a node to the network and another that fires a node. Both of these transformations could potentially be applied to the same input model, making the choice of which atomic action to take non-deterministic. When computing the execution traces of this language, a separate trace needs to be created for each of these choices to reflect the non-determinism.

Another challenge and different source of non-determinism in the semantics is found in the pattern matching of the transformation rules. The module that computes the execution traces needs to know which choices in the pattern matching are non-deterministic so that a new trace can be computed for each such choice. The difficulty is specifying this non-determinism in a way that is modeling language agnostic.

A third challenge that was addressed is how to store execution traces efficiently. The static structure of a model that does not change during execution can represent a significant portion of large models. Instead, a rather small portion representing the current state of the model may be the only thing that changes between execution steps. In these cases, a naive mechanism for storing the individual execution steps that does not leverage this knowledge can introduce significant storage overhead.

Contributions

This dissertation makes the following contributions. The first is a unified framework in which Statechart models of different semantic variants can be defined, simulated and verified. The framework is integrated with an analysis tool to perform verification. The key idea is that the user describes only the structure of a Statechart model. The structure is then automatically translated into equivalent Java code, and the semantics are selected from a set of pluggable Java components. Components implementing the semantics of three different variants of Statecharts were defined: Matlab/Stateflow, UML and Rhapsody. By decoupling the structure from the semantics, a single model can be easily executed using multiple semantics, and a system comprised of interacting models using different semantics can be simulated and verified in a single environment. The interaction between models is captured through the input/output interface of the models.

To perform analysis, the framework is integrated with Java Pathfinder [14], a software model checker, along with Symbolic Pathfinder [15], its symbolic execution engine. Symbolic execution allows both test-vector generation and reachability analysis, which can be difficult with reactive systems that read inputs from their environment.

Additionally, an initial port of the framework to the C# language was done so that the symbolic execution engine Pex [16] could be evaluated on the models.

A lightweight method to specify properties that should be monitored was also implemented. The method is based on the property specification pattern system described in [10]. Properties are specified through an intuitive user interface, from which Java code to monitor these properties is generated. This allows the user to specify a wide-range of commonly occurring properties very quickly.

The second major contribution is an extension to Formula that calculates execution traces of models using the behavioral semantics, which are defined as described above. The module to calculate execution traces consists of three components. The first is a component that applies all applicable transformations to an input model at a given step and creates a separate trace for each such application. The second component is used to create a separate trace for each non-deterministic choice of the input parameters that are passed to a transformation. This makes non-determinism inside a single execution step explicit to the trace computing module. The third component is a tool that stores the execution traces efficiently by computing and storing only the differences between consecutive steps in a trace when possible. Additionally, a prototype tool for visualizing the execution traces was also developed.

Outline

The rest of this dissertation is structured as follows. Background material on modeling languages, semantics, analysis and Statecharts is presented in Chapter II. The Statechart analysis framework is described in Chapter III. The analysis extensions to Formula are presented in Chapter IV. Chapter V concludes and give directions for future extensions.

CHAPTER II

BACKGROUND

This chapter presents background material on modeling languages, semantics, formal analysis and Statecharts. Unless stated otherwise, the term modeling language refers to a software-system modeling language.

Modeling languages

As defined in [3], a model is a formal structure that represents selected aspects of an engineering artifact and its environment. The phrase "selected aspects" means that a model is an *abstraction* of some real-world object. Models can be used for many different purposes, including design, validation, testing, simulation and analysis.

In order to build a model, a *modeling language* is used. A modeling language consists of the following elements.

1. A set of concepts and associated attributes, along with the relationships between those concepts. This is referred to as the *abstract syntax* of the modeling language.

2. A set of rules defining the notations used to express models. This is referred to as the *concrete syntax*.

3. A set of rules defining what a model means. This is referred to as the *semantics* of the model.

Traditional programming languages are sometimes divided into just two components: syntax and semantics. In this view, the first two items in the list above can be thought of as the syntax.

Models can provide numerous advantages in software development. Besides providing an abstraction mechanism that can alleviate the fundamental limiting factors of human cognition [17], models are also helpful when a full system description is too complex for even a computer to reason about [18]. Models can also be used for a variety of other purposes, including helping developers and customers communicate, test-case generation or derivation of the developed system [19].

Modeling languages can be general, in which case they can be used to describe a wide variety of systems, or specific, in which case they contain concepts for describing a narrow field of systems. A modeling language of the second type is referred to as a domain-specific modeling language (DSML). In either case, the first step in creating a modeling language is to define the syntax, which involves two steps.

1. Defining the abstract syntax (including well-formedness rules).

2. Defining the concrete syntax (either textual or visual).

The abstract syntax is defined using a meta-language: a language for describing languages. Examples of such meta-languages include MetaGME [3], MOF [20] and UML Class Diagrams [21].

Additionally, the abstract syntax often requires the use of another language to specify well-formedness constraints on models. This is a difference between modeling languages and traditional programming languages. With traditional programming languages, context-free grammars [22] are sufficient to express the allowable constructs of the language, while non-context free constraints (e.g., ensuring a variable is declared before it is used) are left to the semantic analysis phase. One advantage of using modeling languages, especially DSMLs, is that the syntax can restrict the allowed constructs and prune away invalid models without relying on a separate semantic analysis phase to guide users away from modeling inconsistencies.

The difficulty is that meta-languages such as UML Class Diagrams do not offer sufficient expressiveness to check all of these types of constraints. For example, suppose one wishes to define a modeling language for directed graphs. In addition to the concept of nodes and edges, a constraint that forbids cycles in the graph may be desired. That is, it should not be possible to start at any node in the model and, following outgoing edges, reach the original node. This constraint is not expressible in MetaGME, MOF or UML Class Diagrams without the use of an additional constraint language.

A common language for encoding additional well-formedness rules is the Object-Constraint Language (OCL) [23]. However, it has been argued that OCL is not an ideal solution to this problem for various reasons. An alternative solution which also provides analysis over the structural semantics of a language is described in [6].

The concrete syntax provides a way to render a model, either for human users or for use by a computer. Traditional programming languages usually have a textual definition, while modeling languages can be either textual or visual.

Semantics

The term "semantics" refers to the meaning of a language. The purpose of a semantics is to give meaning to the legal sentences of a language. Stated differently, a semantic definition of a language provides an interpretation that provides legal sentences of the language with a meaning. How this meaning is assigned will be discussed shortly.

There has been much confusion regarding the meaning of the term "semantics" as applied to modeling languages; a thorough description and background is given in [24]. Some of the reasons listed there include the misconceptions that (1) the syntax (abstract or concrete) provides the semantics, (2) the semantics must be executable, and (3) semantics can be defined by the meaning of individual constructs rather than the entire language.

One term in particular that sometimes causes confusion is *structural semantics*. Structural semantics describe the meaning of models in terms of the structure of model instances [25]. Hence, structural semantics refer to the meaning of models in terms of the abstract-syntax and well-formedness rules and is a specialization of the more general term semantics. Structural semantics can also be thought of as a decision procedure that examines a model and determines whether the model's structure is correct with respect to the rules that define allowable model structure.

Semantics assign a meaning to a language by providing two things.

1. A semantic domain that is well-defined and well-understood.

2. A semantic mapping from the syntactic elements of the language to the semantic domain.

As an example, consider a traditional programming language, such as C [26]. The ultimate "meaning" of an individual C program is defined by the execution of a set of machine level instructions by some particular computing platform. The semantic domain here consists of a set of machine level instructions and the platform that executes these machine level instructions. The semantic mapping is defined by a compiler and assembler that translate programs written in C into the set of machine level instructions.

In order to describe and reason about languages at a higher level, broad methods for defining semantics of a language have been proposed, each with a different set of merits. The next section describes these broad frameworks. Most approaches to defining semantics fall within one of these categories or a combination of them.

There are three broad categories of semantic description languages.

1. *Denotational semantics* [27] are based on mathematical foundations; they map programs to mathematical functions.

2. *Operational semantics* [28] are based on the concept of an abstract machine; they map programs to this abstract machine.

3. *Axiomatic semantics* [29] are based on treating properties of programs as assertions (predicates); programs transform these assertions.

A description of each of these is given presently, along with examples.

Denotational Semantics

Denotational semantics [27], [30] are a formal method for defining the semantics of programming languages in terms of mathematical functions. A mathematical function is defined which maps the syntax of the program to its semantic value, or *denotation*. That is, it maps a program to the function *denoted*. The denotations are specified using lambda notation, a variant of the lambda calculus [31] that handles data-types.

The semantic functions are defined compositionally. First, a denotation for each basis element in the syntactic category is defined. The semantic functions for composite elements of the syntax are then built by applying the functions to the immediate constituents of the composite element.

A complete description of denotational semantics is far beyond the scope of this survey; a good treatment with examples can be found in [27].

Denotational semantics have been used to aid language designers by serving as a way to specify the language unambiguously. They have also been used to reason about programs and even generate compilers [32], [33]. However, the compilers generated from denotational description of a language are almost always too slow to be used in practice, and often serve as a proof of concept or prototype.

Operational Semantics

Operational semantics [34] regards a program as running on an abstract machine. The semantics is defined by providing a translation from the programming language to the abstract machine, along with rules governing the execution of this abstract machine. The actual abstract machine can be high-level, which makes translation easy, or low-level, which makes precise reasoning easier.

There are different approaches to operational semantics. The following sections describe two: *structural operational semantics* [28] and *abstract state machines* [35].

Structural Operational Semantics

Structural operational semantics (SOS) [28], also called small-step operational semantics, is concerned with describing how the individual steps of a program's computations take place. It provides this description using *transition systems*, which are a structure $< \Gamma, \rightarrow >$ where Γ is a set of elements, called configurations, and $\rightarrow \subseteq R \times R$ is the transition relation. The transition relation is given by defining a set of axioms and inference rules.

Examples of languages that have been described using SOS include a large sublanguage of Java [36] and several variants of Statecharts [37]. SOS has been used in program analysis in [38] and in program verification in [39].

Abstract State Machines

Abstract state machines (ASM) [35] were originally called evolving algebras [40]. They treat the static structure of a language as terms over an arbitrary algebra (i.e., a set of elements along with operations on those elements) and capture the dynamic behavior by rules which may update the operations; thus the algebra *evolves* because the operations may change with time. The abstract state of an ASM is a set of arbitrary data structures, and the operations are those that can be performed by a finite-state machine over these arbitrary data structures.

ASMs began as an attempt to bridge the gap between formal models of computation and practical specification methods. As a specification language, ASMs offer some advantages over other formal methods and semantic description languages. ASM programs use a simple syntax that is almost like pseudo-code. This is in contrast to methods such as denotational semantics (described above) which use a complex syntax. ASM specifications are also executable, another added benefit compared to other specification languages that are not directly executable.

Examples of approaches that have used ASM for specifying the semantics of modeling languages include semantic anchoring [25] and the model checking approach [41] in which ASM is used as an intermediate language.

Axiomatic Semantics

The axiomatic approach to defining semantics [42], [29] is based on the view that properties of programs can be viewed as assertions, or *predicates* over a program's data. A predicate is a statement that is either true or false depending on the values of its variables. Thus, a predicate over a computer program's data is true or false depending on the values assigned to the program's data at a given time. Axiomatic semantics view a program as transforming these predicates, or equivalently, as a *predicate transformer*.

A common formalism of axiomatic semantics is called *Hoare logic*. A *Hoare triple*, $P\ \{Q\}\ R$, states the connection between a precondition (P), a program (Q) and a description of the result of its execution (R). A set of axioms describing the logical inferences that can be made using the triples is presented in [29].

One major limitation in the original description found in [29] is that it lacks axioms and rules to deal with goto (jump) statements. Some have argued that this is a reason against such constructs in programming languages [43]. [44] provides details on how to augment Hoare logic to deal with such statements.

Axiomatic semantics are the most abstract of the three methods described. For this reason, they are primarily useful for proving properties and assertions about small programs or algorithms, and are not well-suited for other analysis techniques, such as simulation.

Behavioral Semantics

Behavioral semantics are not considered a broad framework or method for defining semantics. Rather, the term behavioral semantics refers to the semantics of languages and systems that have a behavior that one wishes to study or observe. Behavioral semantics define the dynamic evolution of a system's state along some model of time. For a modeling language, this means that behavioral semantics describe how the state of a model evolves over time. The mechanism used to describe this evolution can be any of the three methods above, although the operational and denotational approach are the most common due to the fact that axiomatic semantics are primarily used for proving properties.

When considering the behavioral semantics for a modeling language, one can come to the incorrect notion that the lack of a behavioral semantics for a particular language implies that the language does not have a semantics. The lack of behavioral semantics does not imply that a language does not have a semantic meaning [24]. Rather, this means that the particular language in question does not have a semantics that is dynamic in nature. A good example is UML Class Diagrams [21]. A UML class diagram is not dynamic in nature, but it does have a meaning that is reflected in its static structure. The same is true for any modeling languages whose meaning is described in terms of its static structure.

In this paper, the focus is on the analysis of modeling languages that have an associated behavioral semantics. For a treatment on the analysis of languages that deal with static structure, see [45] or [46].

Formal analysis methods

Formal methods and automated verification [47] provide ways of ensuring that software is correct with respect to a specification and free from specific types of flaws. These methods can be *static*, in which case they compute compute information about the behavior of a program without having to execute it, or *dynamic*, in which case they analyze the program through an execution.

The following sections describe three methods for analysis. *Abstract interpretation* deals with the formalization of approximation. By approximating the execution of a program, it can provide sound guarantees about program execution efficiently. *Model checking* determines whether a model of a system satisfies a specification. *Theorem proving* attempts to determine if a sentence of a theory is provable using the axioms and inference rules of that theory. Test input generation is also described, because even though it is not a formal analysis technique, it does provide a useful and practical way to reason about code, and recent extensions can help determine reachability properties of code. It is also relevant to modeling languages for *reactive systems* [48] whose dynamic behavior is driven through an interaction with the environment.

Abstract Interpretation

Abstract interpretation [49] is a broad framework in which many formal methods can be framed. The theory of abstract interpretation formalizes the notion of approximation. Mechanical program verification tools are all similar in the sense that they make a choice regarding the approximation of the behavior of a system and then reason over that choice. Hence, they differ only in the choices they must make to cope with undecidability or complexity [50].

Abstract interpretation makes the distinction between *concrete semantics* and *abstract semantics.* The concrete semantics of a program formalizes the set of all possible executions of this program in all possible execution environments. Because this set of all possible executions is not computable, all non-trivial questions about the concrete semantics of a program are undecidable [50].

The abstract semantics of a program are a superset of the concrete program semantics. That is, they are an over-approximation of the actual behavior of the program. The implication is that if the abstract semantics are correct with respect to a specification, then the concrete semantics will also adhere to the specification. For this reason, abstract interpretation is said to provide *sound* approximations. An approximation is sound if its correctness implies the correctness of the original system.

Static analysis methods based on abstract interpretation work by relating *abstract analysis* to program execution. The key concept is that of an *abstract domain*: an abstraction of the concrete semantics in the form of (1) abstract properties, and (2) abstract operations.

The following sections illustrate with some examples of abstract domains.

Numerical Domains

Numerical domains are used to express properties about the numerical values assigned to variables during program execution. In general, the more precise a domain is with respect to the information it reflects about actual program execution, the slower it performs.

One example of a numerical domain is that of intervals. A numeric interval domain can capture information about the range of values that a variable can take during execution. For instance, consider the following C program.

```
int x = 0;    // A
while (x < 5) // B
  x = x + 2;  // C
```

An interval domain applied to this program to analyze the possible values that the variable x can take during program execution would assign the interval [0,0] to location A, the interval [0,6] to location B and the interval [0,4] to location C. A less powerful domain (in the sense that it loses more information than the interval domain) is that of *Signs*. The Signs domain has three values, {Pos, Neg, Zero}, indicating that a variable has a positive, negative or zero-value, respectively. In the example above, the variable x can have the value {Zero} at location A, while it has {Pos, Zero} at locations B and C.

Relational numerical domains express relationships between the values of variables; examples include *difference bound matrices*, *octagons*, *octahedra* and *polyhedra* [47]. Other typical domains are concerned with *shape analysis*; these domains are used for analyzing properties of the heap and pointers. A succinct description of each of these is available in [47].

Tool Support

Automated tool support for static analysis methods based on abstract interpretation has grown significantly in recent years. An abstract interpretation tool developed at INRIA found the error in the software for Ariane 5 rocket [51]. The software model checker BLAST [52] uses abstract interpretation techniques as part of its analysis for C programs. The code contracts library provided by the .NET framework has a general abstract interpretation framework [53]. The Clang compiler also has a static analysis library [54] for analyzing C and Objective-C programs.

Model Checking

Model checking was proposed independently in both [55] and [56]. In its original definition, the word "model" did not refer to an abstraction of the actual system, but was used because the goal was to check whether a Kripke structure M was a *model* for a temporal formula f [18]. Currently, the term generally refers to checking an abstraction of a system. *Software model checking* usually refers to checking implementation level code.

At its core, model checking is a method for performing *state-space exploration*. A state-space is represented by a directed graph in which the nodes represent the system state, and the edges between nodes represent transitions between states. Thus, the state-space gives a description of how a system evolves. A *model checker* takes as input a description of a system in its modeling language, computes the state space, and then explores this state space to determine properties of interest. Properties that can be discovered by model checkers include the presence of deadlocks, dead code, and violations of user-specified assertions. The specific format in which the user specified assertions are written depends on the particular model checker, but temporal logic [48] is a frequently used format.

Model checking has been used extensively in the verification of both hardware [57], [58] and software [59], [60]. Both hardware and software model checking share the fact that they both perform state space exploration, but they differ in that verification systems for the two areas have evolved in slightly different directions, leading to two different sets of tools based on different logics and that use different types of search and optimization algorithms [59]. One of these differences is illustrated with a description of the *state explosion* problem.

The biggest drawback to model checking is the state explosion problem [61]. The problem is that a model is often described implicitly as a synchronized product of several components; when this structure is flattened, it results in a state space that is exponentially larger than the size of the implicit description. Model checkers designed for hardware applications often deal with this problem by using *symbolic* methods to represent the state space as opposed to an explicit representation such as a list or table. The first demonstration of this technique was done in [62], which used binary decision diagrams (introduced in [63]) to represent the state space; this allowed an increase in the practical size of state spaces that could be searched from 10^8 to 10^{20}.

Software model checkers usually deal with the state explosion differently. State explosion for software systems often occurs when checking a concurrent system (i.e., one with multiple processes executing simultaneously). This concurrency is modeled by representing it as the interleaving of the processes, leading to a combinatorial explosion in the state space. One approach to deal with this is called *partial order reduction* [64]. Partial order reduction works by building an automaton to represent the system and property one wishes to verify that is much smaller than the usual product automaton. Partial order reduction eliminates the need to build an automaton that represents the program and thus has a head start over methods that require the automaton representing the state graph of the program to be built first. For a complete description, see [64].

In the event that a model does not satisfy a specification, model checking can produce a diagnostic counterexample execution trace that shows how the model violates the property. This is one of the biggest strengths of the model checking approach and can be invaluable when debugging complex systems [18]. Other advantages of model checking include its speed, lack of need for proofs and its ability to handle partial specifications.

Model checking has been applied extensively to concurrent systems and protocol verification [65]. The later work is especially interesting because it is a *probabilistic* approach. It describes a program called *Supertrace* that takes the full information for each state of the system under analysis and hashes the information to generate the address of a single bit in memory. If that bit is on, then the state has been seen before, otherwise, it has not been seen. The probabilistic nature arises because if two different states hash to the same value, the program will not notice. Model checking has also been applied to systems which are probabilistic in nature [66] as early as the mid-1980s and more recently in [67].

As noted in [9], there is a continuing shift from verifying manually constructed *models* of code to the direct verification of the implementation level code. This is also evidenced in the growing number of software model checkers [68], [69], [70].

Theorem Proving

Theorem proving [71] is a technique to determine whether a sentence of a theory is provable using the axioms and inference rules of the theory. If the sentence can be proven, then it is called a *theorem*. There are two categories of theorem provers: automated and interactive. Automated theorem provers [71] require no interaction from the user, while interactive theorem provers [72] can interact with a user during their execution to receive guidance about how to proceed with a proof.

Theorem provers are often used in program verification frameworks to prove *verification conditions*, logical formulas whose validity implies that the program satisfies the properties under consideration. Because verification frameworks encompass several different areas (i.e., the program, a theorem prover and the framework itself), many tools use intermediate languages to separate the concerns. In these cases, an intermediate language for generating verification conditions can be used to generate the formulas that are given directly as input to theorem provers [73].

A good discussion about the relationship of theorem provers and satisfiability modulo theories (SMT) solvers [74] is found in [75]. There is some overlap between theorem provers and SMT solvers: both check the satisfiability of first-order formulas. The difference is that theorem provers mainly consider plain first-order logic, while

SMT solvers deal mainly with quantifier free problems and usually include several built-in theories (i.e., axioms and rules) that allow problems from different domains to be encoded naturally. For instance, the theory of arrays allows problems about array accesses in a program to be encoded in a natural way, while the theory of bit-vectors allows one to reason precisely about the computer representation of numbers and arithmetic operations on this representation. SMT solvers are also referred to as automatic theorem provers.

When program verification frameworks generate verification conditions, they generate a formula whose validity implies that the program is correct. One way to do this is by first *negating* the original formula and then giving it to the theorem prover, which attempts to find a solution. A solution to this negated formula means that the original program is not correct. Thus, if the theorem prover finds a solution to the verification conditions, this solution can be presented to the user as an instance of why the program is not correct. In the case that the formula is not negated and the theorem prover fails to find a proof, reporting reasons for this failure can be difficult; [71] presents two methods that a theorem prover can use to give feedback to the user.

In comparison to model checking, theorem provers provide a low level way of determining if a logical formula is valid. Theorem provers are often used as the underlying tool to check verification conditions generated by program verification frameworks [73]. In contrast, model checkers compute a state transition graph of a program and perform state space exploration on this graph. Model checkers have the advantage that if they determine that a model does not satisfy a specification, they can generate a diagnostic counterexample trace that shows exactly what lead to the problem. Theorem provers suffer from the drawback that they cannot always give a concise reason for a failed proof attempt, while model checkers suffer from the state explosion problem. One advantage theorem provers have over model checkers is the ability to handle models of arbitrary size. Model checkers are limited in this respect because they can only work with models of finite size.

Test-Case Generation

Although it is not considered a verification technique, test-case generation [76] is a useful method in the design process that enables testing. Testing cannot exhaustively prove the absence of errors in a system like verification methods, but is a useful technique that is widely used in industrial settings.

There are two main issues with which test-case generation is concerned. The first is how to choose a good sample of inputs. The second is how to exercise different branches of a program. The goal is to find sets of inputs that drive a program through different execution paths. For instance, consider the following C function.

```
void test(int x) {
  int y;
  if (x > 0)
    y = 1;
  else
    y = 0;
}
```

The function test has two possible paths of execution: one path is taken if the value of the parameter x is greater than zero (in which case y is assigned a value of 1), and the other path is taken if x is not greater than 0 (in which case y is assigned a value of 0). A test-case generation method applied to the function test above attempts to find values of x which drive the execution of the program down each of these two paths. In this case, a suitable set of assignments would be x = 0 and x = 1.

As the complexity of code increases, manually performing such an analysis becomes infeasible for two reasons. The first is that the set of constraints that bind a certain set of input variables to a particular path grows very large. The second is that the number of variables that have a constraint associated with them can also grow very large. For instance, consider a modified version of the code above.

```
void test(int x) {
  int y, z;
  z = x + 1; // z now depends on a symbolic input variable
  if (z > 0 && x < 1)
    y = 1;
  else
    y = 0;
}
```

A new variable z has been introduced that is assigned a value of x + 1. The two feasible paths through the function now depend on the values of both x and z. In this case, there exists only one value for x that will drive execution through the "if" branch (x = 0). All other values of x drive the execution through the "else" branch. This simple modification to the code requires our analysis to keep track of the variable z because its value depends on an input variable, and also increases the set of constraints that must be solved in order to find input values that exercise both branches of code.

Symbolic Execution

In the simple code listings above, inputs can be easily determined by hand. Symbolic execution [77] was introduced in 1976 to provide an automated way of performing this analysis. The goal is to allow a computer to analyze code for interesting sets of inputs that exercise as much of the code as possible. Symbolic execution works much like a normal program execution, except that the input variables are not assigned a concrete value. Instead, they are treated as symbolic variables. Program execution proceeds as normal until an instruction involving an input variable is encountered. This instruction can be either an expression involving an input variable or a conditional statement involving an input variable. In the first case, the symbolic value assigned to the input variable is used in the evaluation of the expression, which may entail assigning a symbolic value to variables that are not explicit inputs or attaching additional constraints to an input variable. In the second case, a conditional statement (e.g., an "if" statement) which depends on the value of an input variable is evaluated. Symbolic execution "forks" its execution and attempts to perform both branches of the conditional statement. It does this by determining for each branch whether the current set of constraints on the variables used in the conditional are satisfiable.

When it was introduced in the late 1970s, symbolic execution was intractable for analyzing large, complex code because constraint solvers were not powerful enough to solve large sets of constraints quickly. In the past ten years, symbolic execution has become feasible for analyzing complex code, largely due to the performance improvements of constraint solvers. In addition, tool support has greatly matured in recent years [68], [16].

Test-case generation is especially useful for analyzing reactive systems [48] that take input from their environment and perform computation based on this input. Test-case generation can be used in these cases to find inputs that an environment could provide to a system that would cause invalid behavior.

Statecharts

Statecharts were introduced by Harel in [12] as an extension of traditional finite-state machines (FSMs) to deal with reactive systems [48]. Intuitively, Statecharts extend FSMs with three major concepts:

1. Depth: states can be nested.

2. Orthogonality: multiple states can be active at one time.

3. Broadcast communication: states can send messages to other sets of states.

Since their original introduction, numerous dialects of Statecharts, each with their own particular semantics, have been proposed. A full description of any of these semantic variants is too lengthy to present here; for a survey of the various dialects, see [78]. Here, a small description of the basic semantics is presented which each variant extends and modifies in its own way, followed by an intuitive example that should give a feel for the language.

Statecharts operate in a series of discrete *steps*. At the beginning of a step, inputs and events can be read from the environment. During a step, the hierarchy of the current state configuration is traversed and transitions can be *fired* if they are enabled. A transition is enabled if its triggering event is the same as the current input event and if its guard condition, a predicate over the data of the Statechart and current state configuration, is true. A transition can have associated actions that are performed when it is fired or tested to see if it is enabled. The firing of transitions can cause states to be exited and others to be entered, each of which may have associated actions. The exiting and entering of states causes the state configuration to be updated. At the end of a step, outputs can be emitted to the environment.

Figure 1 shows an example of a Statechart with two *states* labeled A and B, and one input variable, x, an integer. There are two *transitions*, one from A to B *guarded* by the condition $x > 0$, and another from B to A guarded by the condition $x < 0$; in order for a transition to be valid, its guard condition must evaluate to true. Chart execution works in the following way. When the chart is initially activated, State A is entered. Execution then proceeds in a series of discrete *steps*. At the beginning of each step, the value of the input variable x is read from the environment. If the

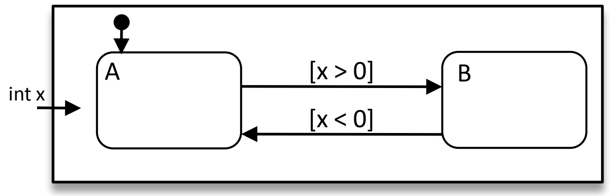


Figure 1: An example Statechart.

Statechart is in State A and the value of x read from the environment is greater than 0, then the transition from A to B is taken, State A is *exited*, and State B is *entered*. If the Statechart is in State B and the value of x read from the environment is less than 0, then the transition from B to A is taken, State B is *exited*, and State A is *entered*.

Statecharts are used to describe a class of systems called reactive systems [48]. A reactive system is one which maintains an ongoing interaction with its environment. During an execution step, a Statechart model reads inputs from its environment, updates its state configuration and emit outputs back to the environment. Most Statechart variants are based on the *clocked synchronous* model of computation: during each tick of a global clock, all inputs are considered, and updating the state configuration and outputting events to the environment is assumed to take zero time.

Given a modeling language for Statecharts and a behavioral semantics that reflects the intuitive description of dynamic behavior given above, one can ask questions about Statecharts such as the following.

- Is a certain state reachable?
- Is it always the case that a certain state is reachable *after* the occurrence of a given condition?

The first question is one of *reachability*. The second question is a *specification* against which a Statechart can be checked for conformance: does the Statechart satisfy the specification? Such specifications can be described precisely in languages such as temporal logic [48] and the Statechart checked for conformance.

CHAPTER III

STATECHART ANALYSIS FRAMEWORK

This chapter describes the Statechart analysis framework. This work was motivated by a real-world problem faced by engineers at NASA. Distributed teams there use different variants of Statecharts to describe interacting pieces of a single system. The differences between the variants are primarily in the rules describing the dynamic behavior of a model, but there are small structural differences as well. Although the differences between the variants are slight, they are crucial and can result in a single model behaving very differently across Statechart variants.

One implication of these differences is that a single model can have a different meaning to different teams. In order to verify that a model has the same dynamic behavior across variants, this property must be checked using each different execution semantics. Without automated support, this can become a laborious task.

Using different variants also complicates system integration when there is communication between models written by two different teams. In this case, none of the environments used to create the individual models can be used to simulate the entire system due to the fact that each tool implements only a single execution semantics. For example, the Matlab/Stateflow environment cannot be used to simulate a Statechart model with the execution semantics used by the Rhapsody modeling tool, nor can the Rhapsody tool simulate a model with the execution semantics of Matlab/Stateflow. This makes verification difficult to perform at the model-level because of the lack of a single environment in which all of the different models can be executed simultaneously.

To address these limitations, a unified environment for Statecharts was designed and implemented. This environment allows multiple, interacting Statechart models, each using different execution semantics, to be simulated and verified. The key point of the approach is that the structure of a model is decoupled from its execution. This allows a model's structure to be defined independently of its behavior. Behavioral semantics for different Statechart variants were defined as Java modules. To use the framework, the structure of a model is first translated into equivalent Java code. This structure code is then combined with a semantic module to provide execution. In this way, a single model can be simulated with different execution semantics very easily: simply plug a different semantic module into the structure code. The approach also provides a single, unified environment in which interacting models executing under different semantics can be simulated and verified.

One related approach is the template semantics described in [79]. The template approach describes the semantics of a particular model-based notation, or in this case, Statechart notation, using parameterized templates. This allows the differences in behavior between the different variants, such as the state hierarchy traversal order for transition paths, to be captured. The description of the semantics given by the templates can potentially be used by a code generator to automatically generate automated tools to analyze the behavior of models that use the semantics. According to [79], for instance, given a particular template-based semantics, an analysis tool that answers reachability questions about models using these semantics could be generated. Cast in this light, the unified framework described in this chapter can be seen as an instance of three such analysis tools: one for analyzing Statecharts that use Rhapsody semantics, one for analyzing Statecharts that use the Matlab/Stateflow semantics and one for analyzing Statecharts that use UML semantics.

This work is also concerned with how model properties that one wishes to verify can be specified and checked. Research has shown that one impediment to more widespread application of formal methods to real-world systems is the cumbersome nature of property specification [10]. For this reason, a property specification method based on the pattern system described in [10] was built. The key idea is that the most commonly occurring properties used in specifications come from a relatively small set which can be divided into two parameterizable pieces. This method allows the user to specify these two pieces through an intuitive interface and automatically translates these into Java code that is used with the execution engine to provide property checking.

To perform verification, the unified Statechart framework is integrated with Java Pathfinder (JPF), a software model checker for Java, as well as its symbolic execution module, Symbolic Pathfinder (SPF). JPF is implemented as a backtrackable Java Virtual Machine that can check properties such as arithmetic overflows, unchecked exceptions and race conditions between threads. The SPF module implements symbolic execution over Java bytecode and can perform test-vector generation.

Symbolic execution (Chapter II) can be used to find inputs that an environment could provide to a Statechart model that would cause erroneous behavior, and is a good choice to analyze the behavior of a Statechart model for at least two reasons.

- The inputs may range over large domains, such as the reals.
- The inputs at a given step may impact the subsequent behavior of the system.

The challenge is to choose input values that drive the system through distinct sequences of states with the two points above in mind. This is different than a simple search problem because in a typical search problem, a single value is chosen at each step. If the value chosen at a given step does not terminate at the desired state, then the system backtracks to a previous step and selects a new value. If the search values range over an infinite domain, such as the set of real numbers, a search that selects one value at a time can be very inefficient.

The second point above means that at a given step, input values to a Statechart can be stored internally by the Statechart and used to control behavior at a later

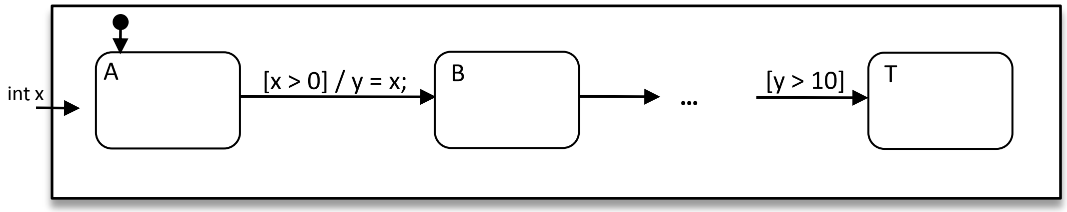


Figure 2: Example in which simple search is inefficient.

time. Consider the example in Figure 2. Assume that x is an integer input, y is an internal integer variable, and that the goal is to find a sequence of input values to x so that starting from the initial state A, the state T is eventually reached (the "..." in the Figure indicates that there is some number of states in between B and T). A simple search might look at the guard condition on the transition from A to B and correctly determine that any value of x greater than zero will enable the transition. However, the subtlety here is that this input value is stored in the value of y, which is not taken into consideration by a simple search. The value of y is not used until much later, where it guards the final transition to state T. From a simple search's point of view, picking any value for the input x during the first step causes the search to proceed to a large depth. However, state T will not be reached until an initial value greater than 10 is chosen for x. Symbolic execution overcomes this problem by not assigning a concrete input to x initially; rather it attaches to x a constraint saying that it is greater than 0. It also uses this constraint when assigning the value of x to y as part of the transition action. When the guard of the transition to T is tested, the symbolic constraint attached to y is used, and in order for the guard condition to evaluate to true, an additional constraint is attached to y, which states that it should also be greater than 10. A constraint solver is then used to check whether the conjunction of the two constraints $y > 0$ and $y > 10$ is satisfiable.

Java was chosen as the language in which the framework is implemented for two reasons. First, the variants of Statecharts, especially Stateflow, have large data and action languages. The action language is used to perform actions during model execution, such as when a transition is taken or a state is entered. Most of the features of the action language are also found directly in Java. Using a simpler language in which features of the action language do not have a corresponding concept would have required an non-native encoding.

The second reason Java was chosen is because there is a powerful model checker for it, Java Pathfinder. While this particular case study is concerned with how to analyze Statecharts, the more general question of interest is how modeling languages can be assigned semantics in a way that makes them amenable to analysis. Part of the goal of this work is to see how well-suited Java is to this task.

One alternative to this approach of analyzing Statecharts is a direct symbolic encoding, which works in the following way. First, a symbolic method is chosen. For instance, a SAT based encoding [80] represents each relevant feature of the language as a propositional variable, while an SMT based encoding [74] can use both propositional variables and additional theories such as linear arithmetic. Second, a translation from the Statechart model into the symbolic representation is performed. Third, the semantics of the particular Statechart variant are used to encode additional symbolic clauses representing the state of the Statechart model as successive steps are performed. All of these symbolic clauses are then given as input to a solver. A satisfying solution to these clauses is interpreted as a way to drive the initial model to a particular state.

While a direct symbolic encoding can be more efficient than defining the semantics in Java and using JPF and SPF to analyze the models, this approach was not taken for two main reasons. First, the ability to simulate a model with different semantic variants is important. The interface to the framework allows users to interactively explore a model's behavior. Interaction with a direct symbolic encoding is very difficult. Second, the semantic variants of Statecharts have small details that are sometimes difficult to see when one is first defining them, and an interactive debugger is helpful in uncovering these details. Debugging a direct symbolic encoding is a more difficult task.

The Java method was also chosen over a direct symbolic encoding to investigate how well modern program analysis tools perform when analyzing code that should be interpreted with a different semantics than the underlying language in which it is written. The Java programming language has a well-defined execution semantics, which is defined by the Java language specification [8]. In the case of the Statechart analysis framework, JPF and SPF are analyzing the code, which represents the semantics of the variants of Statecharts, with respect to the semantics of the Java because that is the language in which the semantics are defined. Ideally, the analysis tools could be configured to interpret the code not using the full semantics of Java, but using the semantics of the Statechart variants. That is, the code could be verified on a different level of abstraction than the level in which it is written. This is addressed further in Chapter V.

Overview

An overall view of the framework is shown in Figure 3. The process includes the following steps.

1. Create a Statechart model.

2. Translate into the intermediate language.

3. Generate the Java code representing the structure of a Statechart.

4. Combine the structure code with a semantic module.

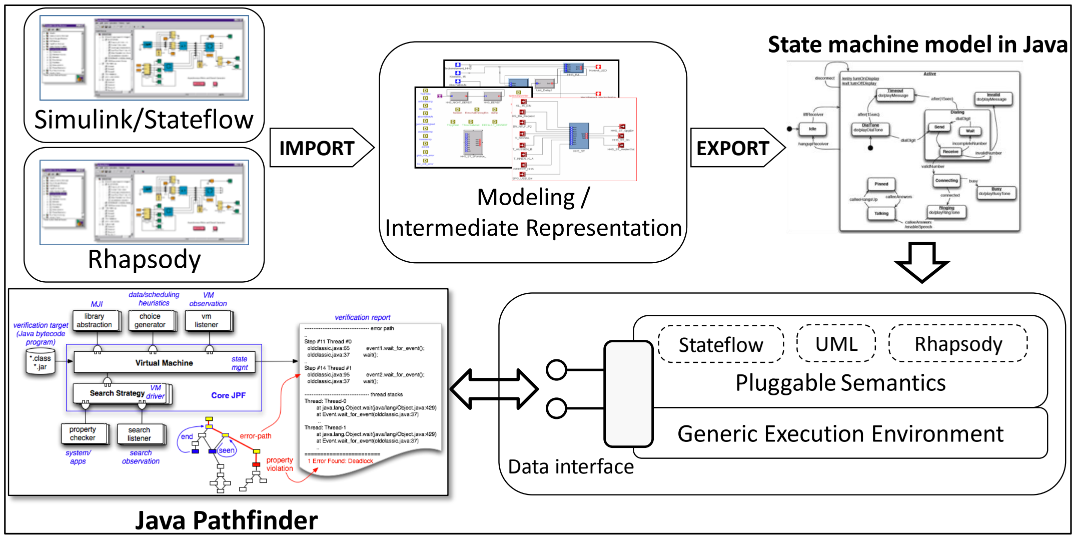


Figure 3: Statechart analysis framework.

5. Analyze with Java Pathfinder.

The first step can be performed with tool support. One method is to use a modeling tool called the Generic Modeling Environment (GME) [3], in which case a Statechart model is created directly in the intermediate language (described below). Another method is to use either the Stateflow or Rhapsody environment for model creation. Translators exist from both of these tools into the intermediate language used by the framework.

The second step, translating the Statechart model into an intermediate representation, uses a language called ESMoL: Embedded Systems Modeling Language [81]. ESMoL was originally designed as an intermediate language into which Matlab/Simulink models can be translated, and from which both Java and C implementation level code can be generated. An intermediate language is needed for the framework for two reasons. First, it simplifies the translation to Java by requiring only one translator targeting Java to be built and maintained. Second, additional information, such as property specifications (described below) can be attached to model elements. The intermediate language provides a place for this additional information to be persisted and analyzed before it is translated into Java.

From the ESMoL intermediate representation, Java code representing the structure of the Statechart model is generated. In addition to translating basic features, such as states and transitions, there are additional features such as graphical functions in Stateflow that make this translation process non-trivial. The process is described in detail later in this chapter.

The fourth step combines the generated structure code with a semantic module. Semantic modules were defined for three variants of Statecharts: Matlab/Stateflow, Rational Rhapsody and UML State Machines. These three variants were chosen because of their popularity.

The Matlab/Stateflow semantics were implemented using the official Stateflow reference manual. The Rhapsody semantics were implemented using [82]. The UML semantics were implemented using the OMG UML Superstructure specification [21]. To further ensure that the correct semantics were implemented, the parametric SOS-style description found in [37] was used as an additional reference.

In addition to running a Statechart with a given semantics, the execution engine also exposes a data interface that is used to set the inputs and read the outputs of a Statechart. The interface enables the inputs and outputs to be accessed in different ways, including access by JPF during analysis. Its design is described later in this chapter.

The last step of the process is to perform analysis using JPF and SPF. There are two parts to performing analysis: (1) specifying properties, and (2) verifying these properties. For the first part, it was mentioned previously that a large number of commonly occurring specification properties can be captured through two parameterizable pieces of data. These two pieces are, (1) a scope, which describes when the property should hold, and (2) a pattern, describing the conditions that should hold while the scope is valid. These two pieces of data can be defined on models directly in the intermediate language, or they can be defined inside the Matlab/Stateflow modeling environment using a custom user interface that was defined. In the later case, they are automatically carried over to the intermediate language. From ESMoL, the properties are transformed into Java code that is used by the execution engine and checked by JPF.

Analysis with JPF can be performed using the Eclipse IDE, which includes a plug-in component for JPF and SPF. JPF and SPF are driven through configuration files in which options and parameters are specified. JPF can check the user specified properties, and SPF can perform test-vector generation. The test-vectors are sequences of inputs to a Statechart that cause it to be driven through a certain sequence of states. State reachability is closely related to the nature of symbolic execution, in which the goal is to find inputs to the system that cause a high-percentage of code to be executed.

The following sections describe individual pieces of the framework in more detail.

Translation of structure

Translating the structure of a Statechart model to corresponding Java code is done in a top-down fashion along the state hierarchy. Starting with the highest level in the state hierarchy, all states at a given level are translated, and then the next lower level in the hierarchy is translated.

Data and scopes

Most Statechart variants allow states to contain internal data variables. The allowed data types depend on the particular variant. Further, access to a variable in a state is scoped in such a way that substates can access the variable as well. Figure 4 shows an example Statechart with two states, *A* and *B*. Note that *A* contains two variables,

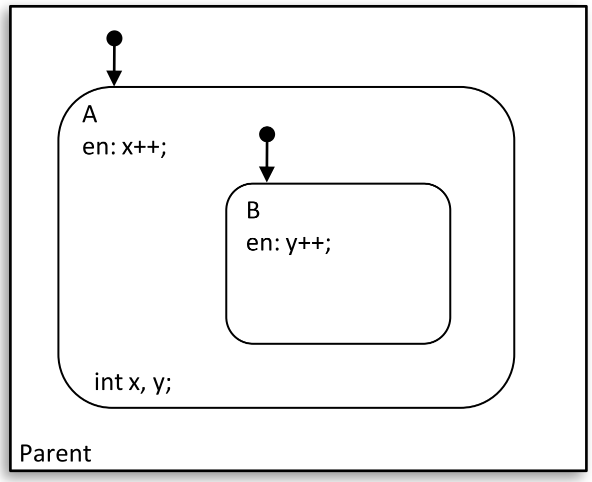


Figure 4: Example Statechart with data used by substate.

x and y, which are used by its substate B. B has access to these variables because they are define in an ancestor state.

Statecharts also allows many behaviors of a model to be customized. For instance, a state has the ability to perform actions at various times during execution, such as when it is entered or exited. In Figure 4, state A has an entry action that increments the value of x, and state B has an exit action that increments the value of y.

This ability to define custom behaviors on model elements influenced the design of the structural Java code to represent Statecharts. First, a set of base classes were defined for the main concepts found across the Statechart variants. A class digram for these main concepts is shown in Figure 5. These main concepts include states, transitions, events, regions and pseudostates. *States* are used to describe a mode of the system. *Transitions* link two objects, such as states, together. A transition begins at a source and ends at a destination. Transitions can have a transition label that describes the circumstances under which the system moves between the source and destination of the transition. *Events* are used to indicate that some event has occurred and can be used to trigger transitions. *Regions* are a concept from the UML

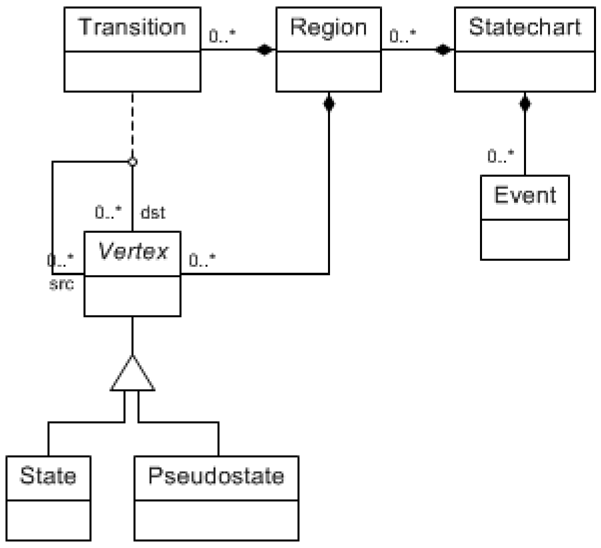
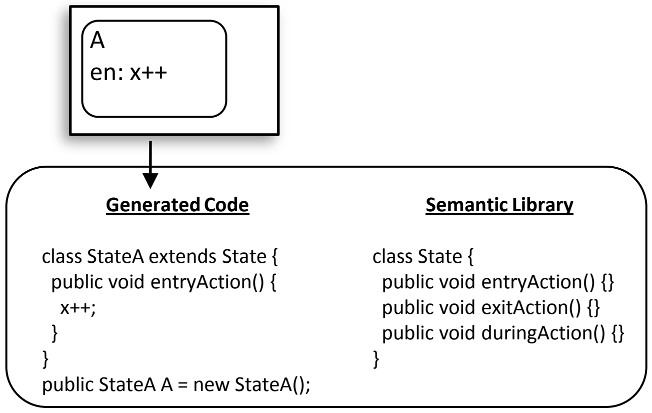


Figure 5: Class diagram of main Statechart features.

semantics and are used to distinguish between states with an exclusive decomposition and states with a parallel decomposition. The idea is that states are contained inside regions, with the constraint that at most one state in a region is part of the active state configuration at any given time. *Pseudostates* refer to objects which are not states, but that can be the source and target of transitions. Junctions are an example of a pseudostate.

The base classes for these main Statechart concepts define methods corresponding to the functionality provided by the concept they represent. For instance, states can perform actions when they are entered, exited and during their execution. Thus, the base class *State* defines three different methods for performing these actions: *entryAction*, *exitAction* and *duringAction*, each of which contains no implementation.

The structure code for a Statechart is generated by extending these base classes and overriding the virtual methods for base functionality with custom behavior when needed. Instances of these extended classes are created and connected to represent the Statechart. For instance, the generated class for state *A* in Figure 6 extends the



```
class StateA extends State {
  public void entryAction() {
    x++;
  }
}
public StateA A = new StateA();
```

```
class State {
  public void entryAction() {}
  public void exitAction() {}
  public void duringAction() {}
}
```

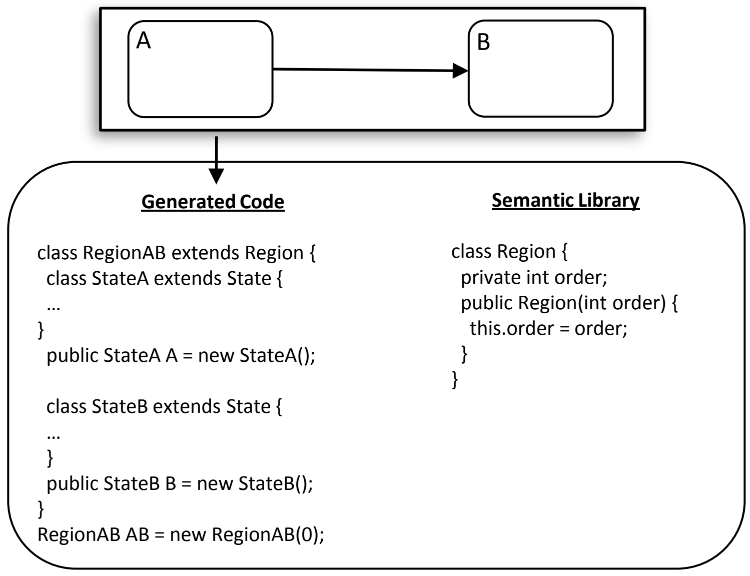
Figure 6: Translation of a basic state.

base class *State* and overrides the *entryAction* method with code to increment the value of x.

Generating custom Java classes that extend the base functionality classes provides a clean solution for handling data and scoping as well. Data defined in a state is generated as a public data variable inside the class corresponding to that state. Scoping is addressed by using nested, inner class. When a child state C is contained hierarchically inside a parent state P, the generated Java class for C is generated inside the generated Java class for P, and the instance of C is created inside the class for P. This allows the Java object representing state C to access all of the public data variables defined in the class for state P and also those defined in the ancestor states of P.

States

Figure 6 shows the translation of a basic state into Java. The state A, shown at the top of the Figure, contains one data member, an integer x, and an entry action that increments the value of x: *en: x++*. The base class, *State*, found in the semantic library, is shown on the right of the Figure and contains three virtual methods: *entryAction*, *exitAction* and *duringAction*. These are overwritten to perform custom



Generated Code

```
class RegionAB extends Region {
  class StateA extends State {
  ...
}
  public StateA A = new StateA();

  class StateB extends State {
  ...
  }
  public StateB B = new StateB();
}
RegionAB AB = new RegionAB(0);
```

Semantic Library

```
class Region {
  private int order;
  public Region(int order) {
    this.order = order;
  }
}
```

Figure 7: Translation of a orthogonal states.

behavior when a state is entered, exited or executed, respectively. The execution engine is responsible for calling these methods at the appropriate time. For instance, when a state is exited, the execution engine will call that state's *exitAction* method to perform the action associated with exiting that state.

The left side of Figure 6 shows the generated code for state A. A unique name, in this case *StateA*, is given to the generated class, which extends the base *State* class. To perform the entry action of state A, the virtual method *entryAction* is overridden to increment the value of x. This method is called by the execution engine whenever this state is entered.

Orthogonal states

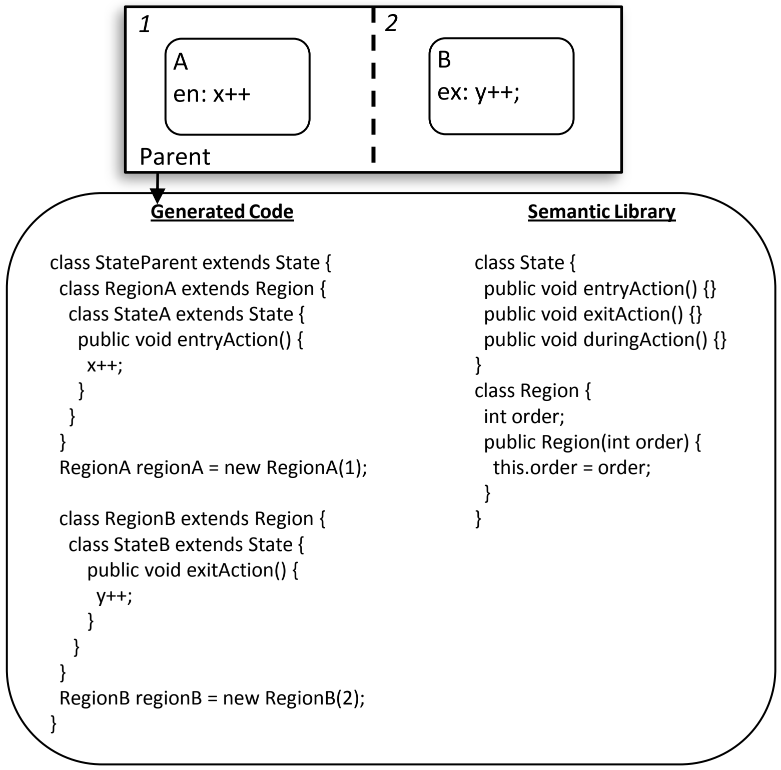
Orthogonal states are states at the same level of hierarchy. The semantics of Statecharts say that a state with an orthogonal decomposition have at most one of their states active at a given time. In order to deal with orthogonal and parallel states, *regions* are used. The idea is that states are contained inside a region, with the condition that at most one state inside each region is active at a given time. Regions are also contained inside states. A state with one region has an orthogonal, or sequential, decomposition, and a state with more than one region has a parallel decomposition. Regions were chosen because they are a concept found directly in the UML State Machine specification and they can be used across the variants to represent orthogonal and parallel states in a modular way.

The top of Figure 7 shows a state with an orthogonal decomposition that contains two substates, *A* and *B*. On the right of the Figure is the base class *Region*, whose constructor takes one parameter telling the activation priority for a region. This priority is used by variants such as Stateflow that allow the order in which parallel states should be entered. Orthogonal states contain only one region, so the priority is not used. The left of Figure 7 shows that state *A* and state *B* are both generated inside the same region, which means that when their parent state is active, exactly one of the two states is active.

Parallel states

A state with a parallel decomposition means that when the state is active, all of its substates are also active. This is reflected in the generated code by placing each parallel state in a separate region. The constructor parameter to the region is used to give the activation order between the parallel states in variants that support this feature.

Figure 8 shows an example of how a parallel state is transformed. The state named *Parent* has a parallel decomposition and two child states, *A* and *B*. The left of the Figure shows the generated code. Note that states *A* and *B* are generated inside different regions, and that the activation order for the regions is preserved in the translation (a lower number is a higher priority).



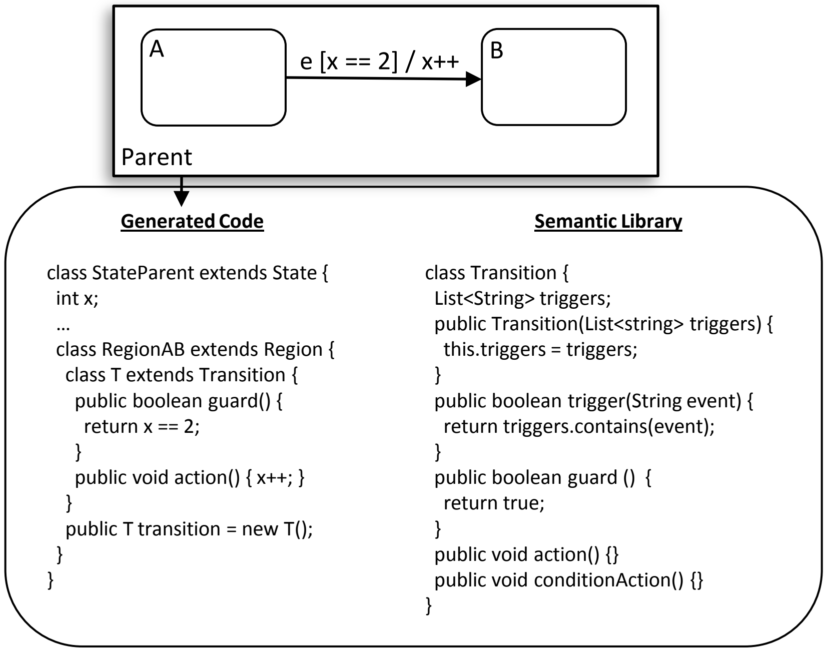
Generated Code

```
class StateParent extends State {
  class RegionA extends Region {
    class StateA extends State {
      public void entryAction() {
        x++;
      }
    }
  }
  RegionA regionA = new RegionA(1);

  class RegionB extends Region {
    class StateB extends State {
        public void exitAction() {
          y++;
        }
      }
  }
  RegionB regionB = new RegionB(2);
}
```

Semantic Library

```
class State {
  public void entryAction() {}
  public void exitAction() {}
  public void duringAction() {}
}
class Region {
  int order;
  public Region(int order) {
    this.order = order;
  }
}
```

Figure 8: Translation of parallel states.



Generated Code

```
class StateParent extends State {
  int x;
  ...
  class RegionAB extends Region {
    class T extends Transition {
      public boolean guard() {
        return x == 2;
      }
      public void action() { x++; }
    }
    public T transition = new T();
  }
}
```

Semantic Library

```
class Transition {
  List<String> triggers;
  public Transition(List<string> triggers) {
    this.triggers = triggers;
  }
  public boolean trigger(String event) {
    return triggers.contains(event);
  }
  public boolean guard ()  {
    return true;
  }
  public void action() {}
  public void conditionAction() {}
}
```

Figure 9: Translation of a transition.

Transitions

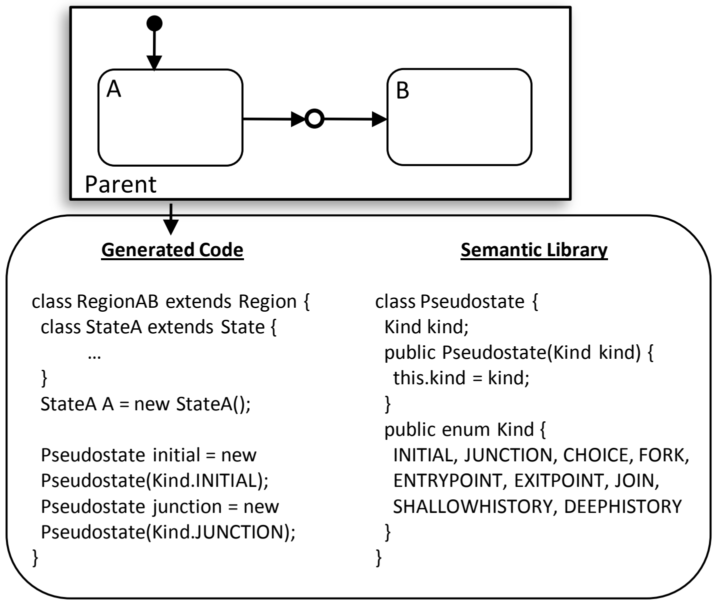
Transitions can have multiple triggering events, an optional guard which is a predicate evaluated over data values and the current state configuration, as well as actions. In order for a transition to be enabled, at least one of its triggering events must be present and its guard must evaluate to true.

The top of Figure 9 shows a transition from state A to state B. Its trigger is the event e, its guard is the condition $x == 2$ and its action increments the value of x (x++). Triggers are implemented as strings, and checking the triggering condition of a transition is done using string comparisons: the base class *Transition* contains a method named *trigger* that takes a string (representing an event) and returns true if this event is a trigger for the transition. Implementing events as strings was done for simplicity, although a class to wrap the strings could have also been used. Figure 9 also shows how the scoping allows the transition to access the variable x defined in state *Parent* through the use of nested inner classes.

Pseudostates

Pseudostates represent objects that can be the source and target of transitions but that are not considered part of the state configuration. Derived classes are not used in the generated code for pseudostates. Instead, an instance of the *Pseudostate* class, defined in the semantic library, is created and its kind is given by an enumeration value.

Figure 10 shows a model with two pseudostates: an initial pseudostate with a transition to state A, and a junction pseudostate in between states A and B. The base class *Pseudostate* is shown on the right, along with the enumeration listing all the possible types of pseudostates. The generated code for the model is shown on the left of the Figure.



```
class RegionAB extends Region {
 class StateA extends State {
       ...
 }
 StateA A = new StateA();

 Pseudostate initial = new
 Pseudostate(Kind.INITIAL);
 Pseudostate junction = new
 Pseudostate(Kind.JUNCTION);
}
```

```
class Pseudostate {
 Kind kind;
 public Pseudostate(Kind kind) {
  this.kind = kind;
 }
 public enum Kind {
  INITIAL, JUNCTION, CHOICE, FORK,
  ENTRYPOINT, EXITPOINT, JOIN,
  SHALLOWHISTORY, DEEPHISTORY
 }
}
```

Figure 10: Translation of a pseudostate.

Stateflow graphical functions

One feature unique to Stateflow that requires special care when translating is the concept of a *graphical function*. A graphical function is a program written with flow graphs using junctions and transitions. Graphical functions can accept arguments and can have multiple return values. A graphical function can be called in the actions of transitions and states. The advantage of graphical functions is that they allow C and Matlab style functions to be defined using junctions and transitions instead of using native code.

Figure 11 shows an example of a graphical function named F that takes two integer arguments, a and b, and returns their difference. Like all graphical functions, Figure 11 consists of only junctions and transitions. The transitions have guards and actions that implement the logic of a difference function. The transition loop in the upper part of the Figure handles the case where a is greater than b, and accumulates the difference by iteratively comparing a to b, decrementing a if it is greater and incrementing x. The transition loop in the lower part of the Figure handles the case

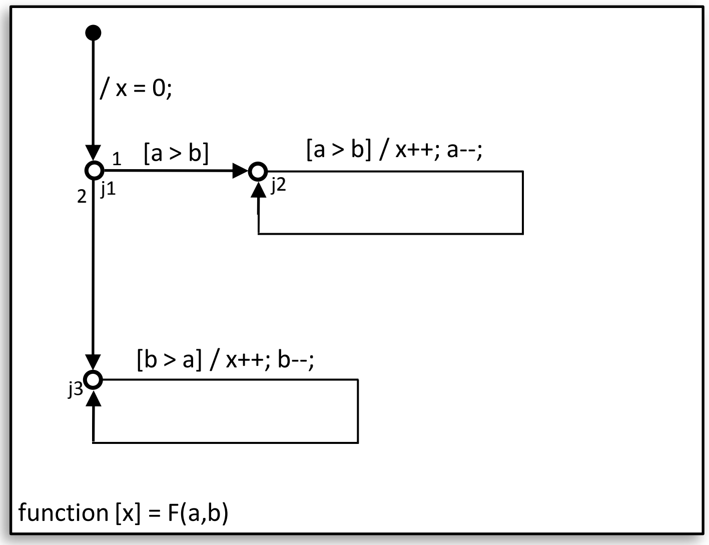


Figure 11: A graphical function state named F that takes two arguments, a and b, and returns their difference. The three junctions are named j1, j2 and j3.

in which b is greater than a in an analogous way. A graphical function terminates when control reaches a junction that has no enabled outgoing transitions.

Figure 12 shows the generated Java code for the graphical function in Figure 11. For each graphical function, a Java class is generated. The function is called using the *init(int a, int b)* method. The first thing the *init* method does is initialize the parameters, which are stored as class instance variables so that all the methods in the class representing the graphical function can access them. Next, the action on the initial transition is performed. In this case, x is initialized to 0. Finally, the method representing the target junction of the initial transition is called, which in this case is the method *j1*. For each junction in a graphical function, a method is generated. This method is called whenever a transition is taken which has the corresponding junction as its destination. The code inside the methods that correspond to junctions contains a conditional for each guard condition on the outgoing transitions of the junction. If this conditional is true, then the action on the corresponding transition is performed and the method representing the junction target of the transition is called.

```
class F extends State {

 /* One instance variable and two inputs to the Function */
 int x, a, b;

 /* The method used to invoke the graphical function */
 public void init(int a, int b) {
  this.a = a;
  this.b = b;

  /* Perform the initial transition action */
  x = 0;

  /* Call the method for the target of the initial transition */
  j1();
 }

 public void j1() {  /* Method for junction j1 */
  if (a > b)
   j2();
  else
   j3();
 }

 public void j2() {  /* Method for junction j2 */
  if (a > b) {
   x++;
   a--;
   j2();
  }
 }

public void j3() {    /* Method for junction j3 */
  if (b > a) {
   x++;
   b--;
   j3();
  }
 }

 public int getReturn1() {  /* Method to get the return value */
  return this.x;
 }
}
```

Figure 12: The generated Java code for the graphical function in Figure 11.

In the example of Figure 12, in the method for junction *j1*, the first conditional is *if(a>b)*, corresponding to the guard on the first outgoing transition from *j1*. If this condition is true, then because the transition contains no action, the method for junction *j2* is called because it is the target of the first outgoing transition of junction *j1*. If the first condition in the method for junction *j1* is not true, then the method for junction *j3* is called because *j3* is the target of the second outgoing transition of *j1*. Execution is complete when a junction with no enabled outgoing transitions is reached. In order to get the return value of the graphical function, the method *getReturn1()* is called. Because a graphical function can have multiple return values, one method is generated for each return value. Whenever a return value is used in a Statechart model, the corresponding generated Java code substitutes a method call to the method to get that particular return value.

Data interface

The data interface allows the generated code representing a model to read input variables from the environment and send outputs to the environment. This feature allows a model to be driven manually by the user, non-deterministically by JPF or symbolically by SPF. The semantic interpreters execute a Statechart by performing the loop shown in Algorithm 1. The execution engine is responsible for reading and setting the inputs at the correct time. The reason for this is that the model mainly acts as a passive structure that is used by the execution engine during execution. Thus, the time at which inputs should be read and sent to the machine is known only by the execution engine.

The data interface is generated using the the same strategy as the generation of the model structure: the core concepts are found in pre-defined base classes, and custom functionality is implemented by over-riding virtual methods and implementing interfaces. The interface to read data must be partially generated because the number

Algorithm 1 Execution loop

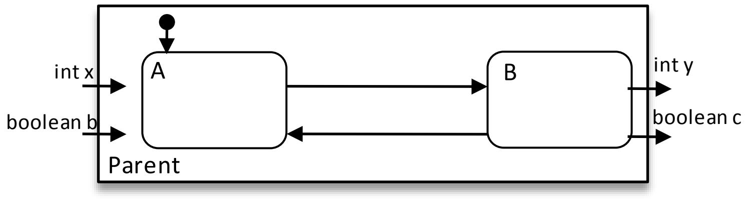
```
while !executionComplete do
    read event
    read inputs
    set model inputs
    step model
    check properties
```

of inputs and their data types is specific to each model, and these inputs must be type-checked and passed in the correct order to the model. In order for the execution engine to call a generic method to perform the reading, type-checking and setting of inputs, it uses an interface. For each individual model, a custom class is generated that implements this interface with functionality specific to the model.

Figure 13 shows an example of a generated data interface and how it is used. At the top of the Figure is a Statechart with two inputs, an integer x and a boolean b. The generated structure code is shown in the middle of the Figure on the right side. The generated class for the top-level state, *Parent*, contains a method called *setInputs*, which is called to set the values of the input variables (marked with a comment *1* in the Figure).

The *IDataReader* interface, used by the execution engine to call the *setInputs* method, is shown at the bottom right of Figure 13. The generated class (*ChartReader*) implementing the *IDataReader* interface for this model is shown in the middle of the Figure on the left side. Notice that the *ChartReader* class contains an instance of the top-level state *ParentState*. This instance allows *ChartReader* to call the *setInputs* method with the correct types for the model's inputs.

The generic execution engine is shown at the bottom of Figure 13 on the left side. In this example, an instance of the *ChartReader* class is passed as the second parameter to the constructor of the execution engine. In the execution engine's *executionLoop* method, the method call *reader.setInput()* results in a call to the generated *ChartReader* class in the middle of Figure 13 on the left side. This method reads two



Generated code

Generated DataReader

```
class ChartReader implements IDataReader {
  private ParentState state;
  private IDataProvider dataProvider;
  public void setInput() {                    // 2
    // read inputs from IDataProvider
    String a = dataProvider.readData();
    String b = dataProvider.readData();
    // parse inputs to correct types
    Int aInt = Integer.parseInt(a);
    boolean bBool = Boolean.parseBoolean(b);
    state.setInputs(aInt, bBool);
  }
}
```

Generated Structure Code

```
class Chart extends Statechart {
  class RegionA extends Region {
    class ParentState extends State { // Highest-level state
      int x; // x and b are inputs
      boolean b;
      public void setInputs(int x, boolean b) {      // 1
        this.x = x;
        this.b = b;
      }
    }
  }
}
```

Base library components

Generic execution engine

```
class Interpreter { // Base interpreter class
  Statechart chart; // Instance of our chart
  IDataReader reader; // Instance of custom reader for chart
  public void executionLoop() {
    reader.readEvent();
    reader.setInput();
    this.step();  // step the Statechart
    checkProperties();
  }
  public Interpreter(Statechart chart, IDataReader reader) {
    this.chart = chart;
    this.reader = reader;
  }
}
```

IDataProvider Interface

```
interface IDataProvider {
  public String readData();
  public boolean hasData();
  public String readEvent();
  public void advance();
}
```

IDataReader Interface

```
interface IDataReader {
  public void setInput();
  public boolean hasData();
  public String readEvent();
  public void writeOutput();
}
```

Figure 13: A Statechart and its generated data interface.

string inputs using an instance of the *IDataProvider* interface. The *IDataProvider* interface is used to allow the string inputs to come from a file, from the user or from the analysis tool. After these two string inputs are read, they are parsed to the correct types, which in this example is an integer and a boolean. Finally, the *setInputs* method is called on the *ParentState* instance to set the model's inputs.

Execution engine

The execution engine is the component responsible for executing a Statechart. Three different execution engines were implemented: one for Rhapsody semantics, one for Stateflow semantics and one for UML semantics.

Algorithm 2 Step algorithm

```
bool traversalDone = false
while !traversalDone do
    level l = selectHierarchyLevel // semantic variant
    if l == endOfHierarchy then
        traversalDone = true
    for all parallelState p in l do
        if computeTransitionPath(p) then
            traversalDone = true // terminate the search if there is a path from any
            state
processTransitions() // semantic variant
```

Algorithm 3 computeTransitionPath(State s)

```
for all Transition t in s.outgoing do
    if t.guard() && t.event() == currentEvent then
        if validTarget(t.target) then
            push(t) // semantic variant
            return true
```

Algorithm 2 shows the *step* method at a high-level. The outer while-loop is responsible for traversing the hierarchy in the correct order for each variant. For instance, Matlab/Stateflow performs the hierarchy traversal top-down, while Rhapsody and UML are both bottom-up. At each level of hierarchy, the algorithm iterates over all of the parallel states found that the current level (the for-all loop in Algorithm 2) and tries to find valid outgoing transition paths from these states (*computeTransitionPath()*).

Most of the details and complexities of the various Statechart semantics are found in the algorithm to compute a transition path (Algorithm 3). A transition path is a sequence of enabled transitions starting from a state in the current state configuration and ending at a valid target. One example of the complexity is the large number of different types of pseudostates found in the Rhapsody semantics, which requires a large number of special cases to handle. The semantic variants also differ in when they perform various actions, such as actions associated with transitions or states. For instance, in Rhapsody, the results of a transition action during a logical step can be seen immediately in that same step, whereas in UML semantics, the results of a transition action are not visible until the next logical time step. Stateflow, on the other hand, can perform transition actions in two different ways. Due to the large number of special cases between the three variants, high-level pseudocode is shown and the places where the semantic variants are encountered are noted.

Algorithm 4 validTarget(Target t)

```
if t is State then
  if t is parallel state then
    for all State s in t do
      if !validTarget(s.getInitial()) then
        return false;
  else
    return true;
else
  process pseudostate // depends on particular kind of pseudostate
```

The *computeTransitionPath(State s)* method shown in Algorithm 3 works in the following way. The outer loop iterates over all of the outgoing transitions of *s*. The first *if* statement tests whether the transition's guard and triggering event are enabled. If they are enabled, then the *validTarget* method is called to test whether there is a valid transition path to the target of the transition. If *validTarget* is true, the transition is added to a stack that stores the transitions on the current transition path. When these are used and processed is dependent on the semantic variant.

The algorithm to check whether a state or pseudostate is a valid target of a transition is shown in Algorithm 4. The first *if* statement checks whether the parameter t is a state or a pseudostate. If it is a state, then the next part depends on the particular semantic variant. In UML, it must be checked whether t is a parallel state, and, if it is a parallel state, then it must be checked that each of t's substates are valid targets. In Stateflow and Rhapsody, whether t is parallel does not have to be checked; rather, a value of true is returned to indicate that the target is valid. The outermost *else* is responsible for the case where the parameter t is a pseudostate. All of the variants deal with pseudostates differently; a comparison can be found in [37].

Property specification

Formal analysis methods provide guarantees about system behavior. The prerequisite to using these methods is a description of relevant system properties in a specification language. The property specification tells the analysis tool which system properties it should check. A large number of languages for describing properties, or property specification languages, exist, including regular expressions and temporal logic, such as LTL and CTL. However, the drawback to using temporal logics for property specification is their steep learning curve for industrial practitioners. Consequently, designers and developers will be less likely to use verification tools if they must devote large amounts of time to learning a specification language.

For this reason, a different method for property specification was integrated into the framework. This approach to specifying properties uses the pattern-based system introduced in [10]. In that work, the authors studied a large body of existing property

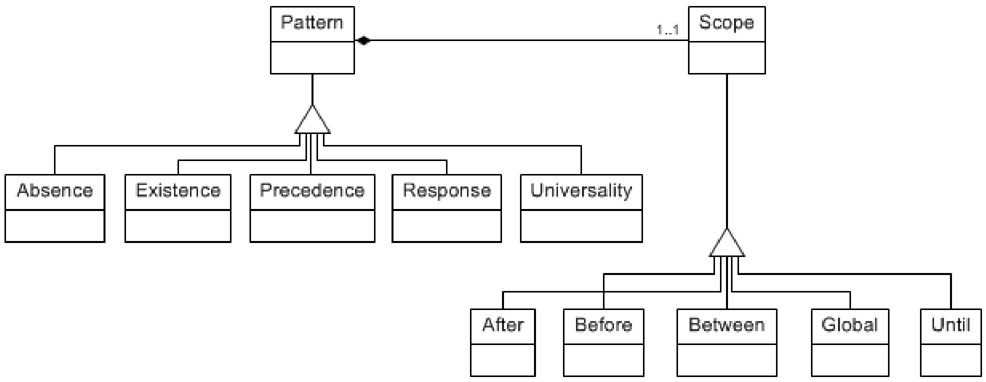


Figure 14: A class diagram for properties represented as scopes and patterns.

specifications and found that the majority of them were instances of a small set of parameterizable patterns: reusable solutions to recurring problems.

Patterns can be entered into the system using a custom interface that was integrated directly into the Simulink/Stateflow environment, or they can be entered directly in the intermediate modeling language. After the parameters have been specified, the framework generates Java code implementing the semantics of the patterns. This code is automatically connected to the execution engine to provide analysis.

To illustrate the pattern-based approach to property specification, consider the property that throughout a system's execution the value of a certain variable should always be greater than zero. There are two basic parts to this property that commonly occur. The first tells *when* the property should hold (in this case, at all times during execution), and the second tells *what* condition should be satisfied during this time (here, the variable should be greater than zero).

A property consists of precisely those two pieces: a *scope* and a *pattern*. The scope defines when a particular property should hold during program execution, and the pattern defines the conditions that must be satisfied. There are five basic kinds of scopes, described below and shown in Figure 15.

- Global: the entire execution.

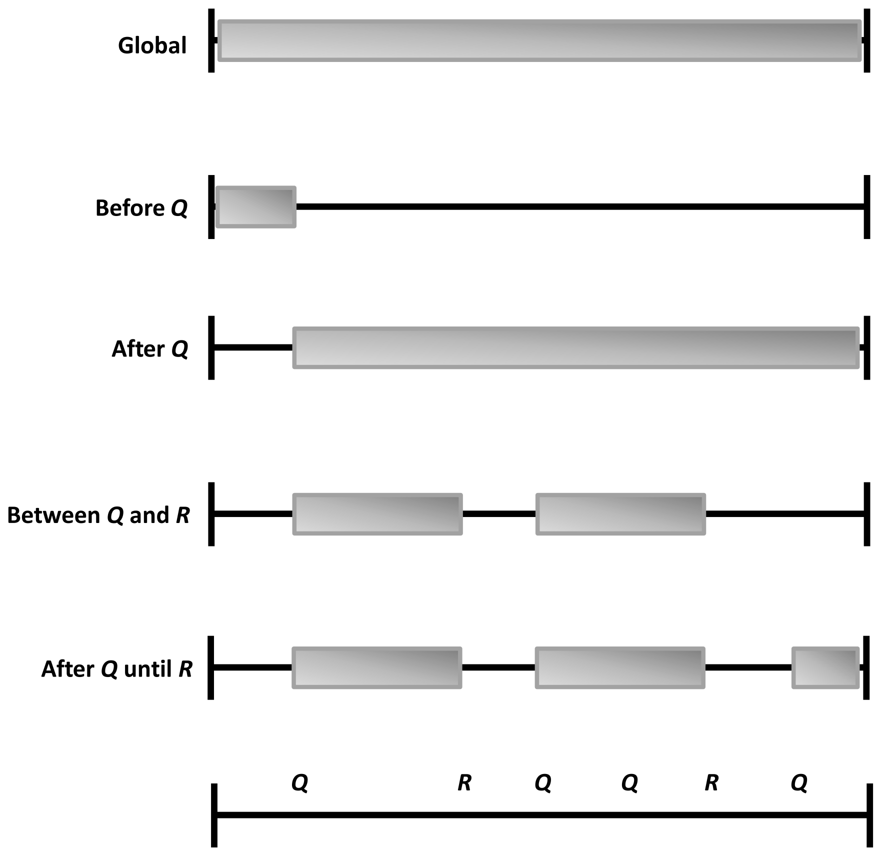


Figure 15: Pattern scopes.

- Before: execution up to a given state.
- After: execution after a state.
- Between: execution from one state to another.
- Until: execution from one state to another even if the second never occurs.

There are two main categories of patterns, occurrence and order, as shown in Figure 16. The occurrence group contains the absence (never true), universality (always true), existence (true at least once) and bounded existence (true for a finite number of times) patterns. The order group contains the response (a state must be

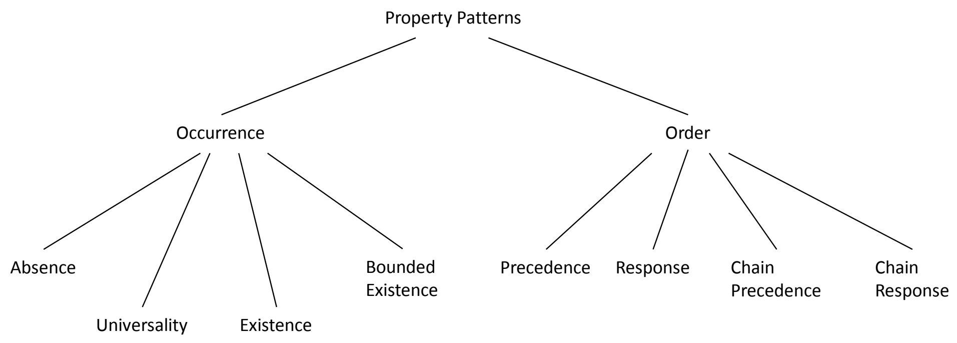


Figure 16: Pattern hierarchy.

followed by another state), precedence (a state must be preceded by another state), chain precedence and chain response patterns.

Scopes

The Java interface representing scopes is shown at the top of Figure 17. The four static integers indicate the status of a scope. An unknown value for a scope means that future information is needed to determine whether the scope should apply. This is used by scopes such as the *Before* scope, in which a scope may or may not be active depending on whether another event happens on a later time. A scope with a status value of post-active means that the scope was active at an earlier time but is currently no longer active. This is used by the *Before* pattern to indicate that the "before" condition has been met and that conditions which depended upon this knowledge in previous steps now know it was true.

The Java class for patterns is shown at the bottom of Figure 17. The *Pattern* class is intended to be extended twice. The first extension is by classes implementing the logic of the concrete patterns listed above. The second extension is by generated code for property specifications for specific Statecharts. The *Pattern* class contains an instance of a class that implements the *Scope* interface, as well as two virtual methods

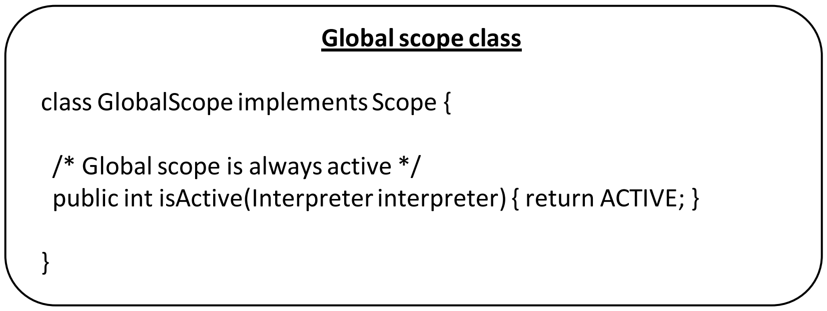
Scope interface

```
interface Scope {
  public static int INACTIVE = 0;
  public static int ACTIVE = 1;
  public static int UNKNOWN = 2;
  public static int POST_ACTIVE = 3;

  /* Returns one of the above values */
  public int isActive(Interpreter interpreter);
}
```

Base Pattern class

```
class Pattern {
  public Scope scope;

  /* Virtual method to be over-written in concrete pattern classes */
  public boolean checkProperty(Interpreter interpreter) { return true; }

  /* Virtual method to be over-written in generated code */
  public boolean checkExpression(Interpreter interpreter) { return false; }
}
```

Figure 17: The Scope interface and base Pattern class.


Global scope class

```
class GlobalScope implements Scope {

  /* Global scope is always active */
  public int isActive(Interpreter interpreter) { return ACTIVE; }

}
```

Figure 18: The Java class for the global scope.

that are intended to be overridden. The *checkProperty* method is overridden by the first extension and is called by the framework to determine if a property has been violated. This method implements the logic of a general pattern. A return value of *true* from the *checkProperty* indicates that the property has been violated. The *checkExpression* method is overridden by the second extension and implements the logic of a pattern instance in the context of a particular Statechart.

Global scope

The global scope is active at all times during the execution of a Statechart. The Java class for the global scope is shown in Figure 18.

After scope

The after scope is active *after* the occurrence of some condition. The Java class for this scope is shown in Figure 19. Initially, this scope is inactive. The condition after whose occurrence the scope becomes active is checked using the *checkAfterExpression* method. As soon as this method returns *true*, the scope is active.

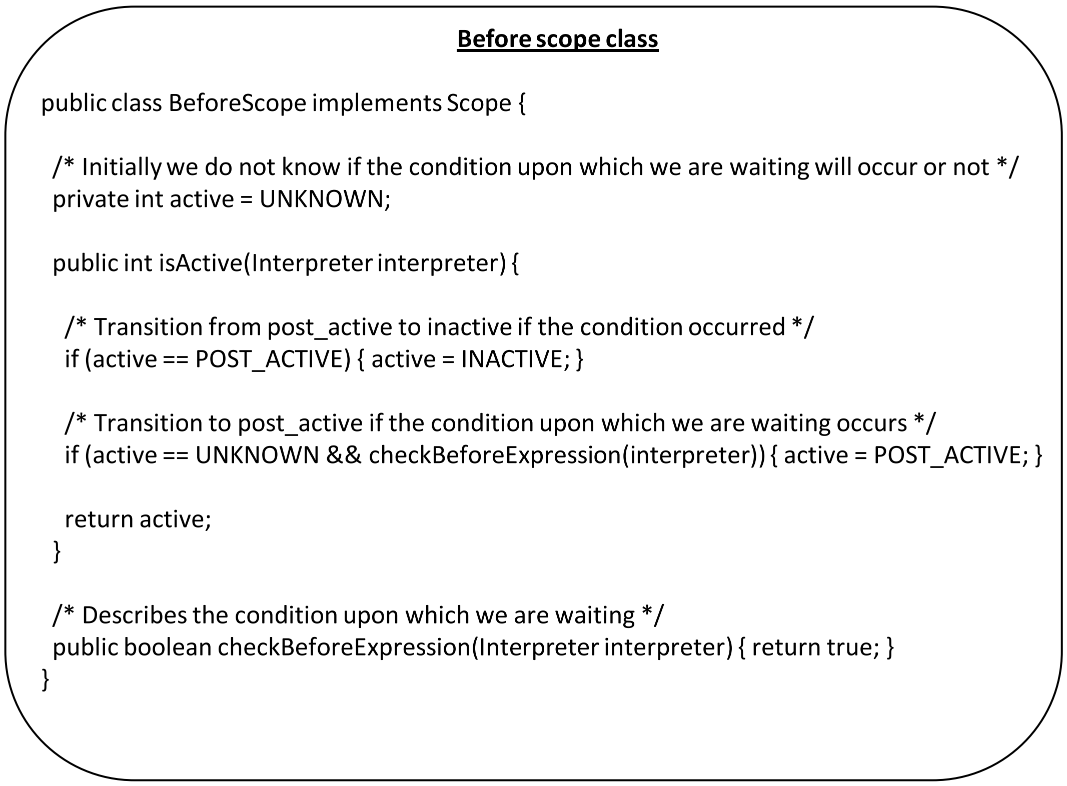
Before scope

The before scope is active *before* the occurrence of a condition that may or may not occur during the execution of a Statechart. If the condition occurs, then the scope

After scope class

```
public class AfterScope implements Scope {

  private int active = INACTIVE;

  public int isActive(Interpreter interpreter) {

    /*We are in scope after the occurrence of the condition upon which we are waiting */
    if (active == INACTIVE && checkAfterExpression(interpreter)) { active = ACTIVE; }

    return active;
  }

  /* Holds the status of the condition upon which we are waiting */
  public boolean checkAfterExpression(Interpreter interpreter) { return true; }
}
```

Figure 19: The Java class for the after scope.



Before scope class

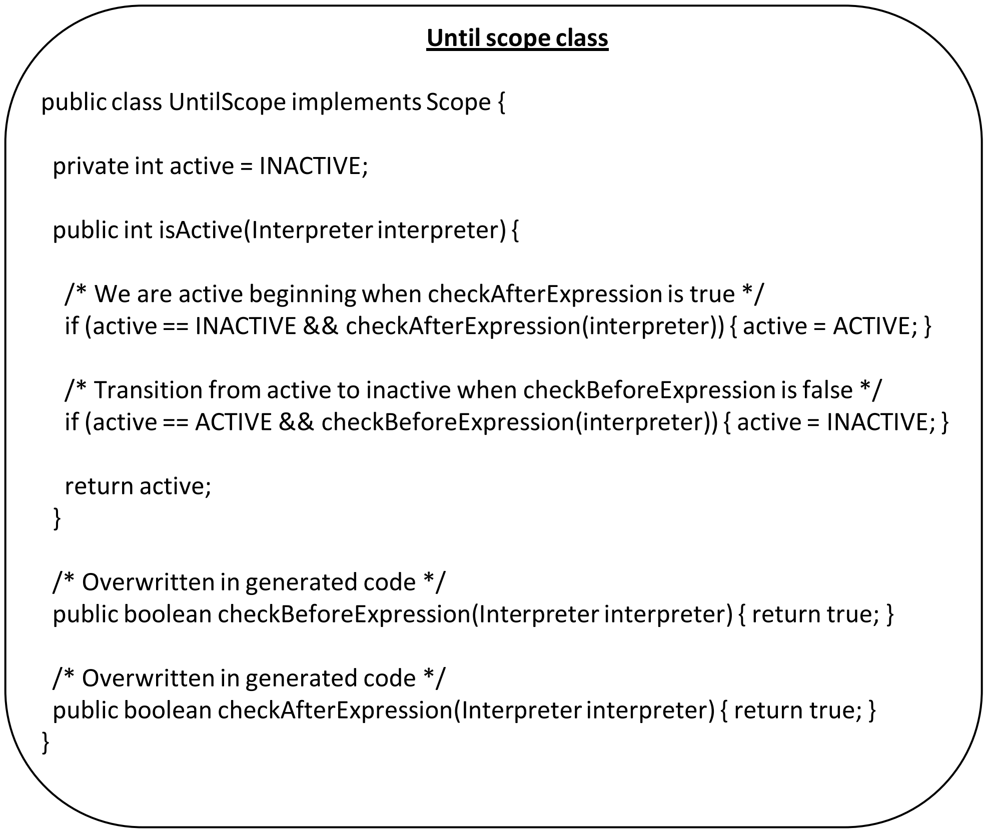
```
public class BeforeScope implements Scope {

  /* Initially we do not know if the condition upon which we are waiting will occur or not */
  private int active = UNKNOWN;

  public int isActive(Interpreter interpreter) {

    /* Transition from post_active to inactive if the condition occurred */
    if (active == POST_ACTIVE) { active = INACTIVE; }

    /* Transition to post_active if the condition upon which we are waiting occurs */
    if (active == UNKNOWN && checkBeforeExpression(interpreter)) { active = POST_ACTIVE; }

    return active;
  }

  /* Describes the condition upon which we are waiting */
  public boolean checkBeforeExpression(Interpreter interpreter) { return true; }
}
```

Figure 20: The Java class for the before scope.

was active at all times prior to the condition's occurrence. If the condition does not occur, then the scope is considered to have never been active. Figure 20 shows the Java class for this scope. Initially, the status of the scope is unknown because it is not known if the condition upon which the property is waiting will occur. The status is set to $POST\_ACTIVE$, meaning that the scope was active at all times previous to the current time, if the condition occurs. The occurrence of the condition is detected by the method call to *checkBeforeExpression*.

Until scope

The until scope is active from the occurrence of one condition to the occurence of another condition, even if the second condition never occurs. The Java class implementing this scope is shown in Figure 21. Initially, that status is inactive. Upon the occurrence of the first condition, which is checked by calling the method *checkAfter-*



Until scope class

```
public class UntilScope implements Scope {

  private int active = INACTIVE;

  public int isActive(Interpreter interpreter) {

    /* We are active beginning when checkAfterExpression is true */
    if (active == INACTIVE && checkAfterExpression(interpreter)) { active = ACTIVE; }

    /* Transition from active to inactive when checkBeforeExpression is false */
    if (active == ACTIVE && checkBeforeExpression(interpreter)) { active = INACTIVE; }

    return active;
  }

  /* Overwritten in generated code */
  public boolean checkBeforeExpression(Interpreter interpreter) { return true; }

  /* Overwritten in generated code */
  public boolean checkAfterExpression(Interpreter interpreter) { return true; }
}
```

Figure 21: The Java class for the until scope.

Expression, the status becomes active. The status remains active until the second condition occurs, which is detected by calling the *checkBeforeExpression*.

Between scope

The between scope is almost identical to the until scope, with the difference that the status of the former is active only if the second condition occurs. Figure 22 shows the Java class implementing the between scope. Initially, the status of the scope is inactive. Upon the occurrence of the first event (detected by calling the method *checkAfterExpression*), the status is set to unknown. The reason for this is that the scope should not be considered active unless the second condition occurs at some point. If the second condition does occur, the status is set to *POST_ACTIVE*, meaning that the status of the scope was active during most recent period during which the status of the scope was listed as unknown. After this, the status is set to inactive if the first condition does not occur. Otherwise, it is set to unknown, meaning that the the status is true if the second condition occurs in the future.

Patterns

The previous sections explained that the base class for patterns, shown at the bottom of Figure 17, is intended to be extended twice: once by modules included with the framework that implement the logic of a general pattern, and a second time by generated code implementing the logic of a patterns in the context of a specific Statechart. This section shows some of the classes implementing the first extension.

Universality pattern

The universality pattern is shown in Figure 23. This pattern is used to represent the requirement that a condition is true at all times while its scope is active. The comments in the code of Figure 23 describe the code intuitively. Notice the two boolean variables *propertyViolated* and *propertyPotentiallyViolated*. These indicate, respectively, whether the pattern has definitely been violated or whether the pattern will be violated if the scope is active. The later is needed to handle scopes such as the until scope which can depend on future information.

Existence pattern

Figure 24 shows the Java class for the existence pattern. This pattern represents the requirement that a condition holds at some point while its scope is active. That is, the condition must occur at least once. The *checkProperty* method begins by checking the status of the scope. If the scope is active or unknown (i.e., it may be active) then the call to *checkExpression* determines if the property has been seen. If so, the

Between scope class

```
public class BetweenScope implements Scope {

  /* Initially inactive */
  private int active = INACTIVE;

  public int isActive(Interpreter interpreter) {

    if (active == POST_ACTIVE) {
      if (checkAfterExpression(interpreter)) {
        active = UNKNOWN;
      } else {
        active = INACTIVE;
      }
    }

    /* Transition from inactive to unknown if checkAfterExpression is true */
    if (active == INACTIVE && checkAfterExpression(interpreter)) { active = UNKNOWN; }

    /* Transition from unknown to post_active it checkBeforeExpression is true */
    if (active == UNKNOWN && checkBeforeExpression(interpreter)) { active = POST_ACTIVE; }

    return active;
  }

  /* Overwritten in generated code */
  public boolean checkBeforeExpression(Interpreter interpreter) { return true; }

  /* Overwritten in generated code */
  public boolean checkAfterExpression(Interpreter interpreter) { return true; }
}
```

Figure 22: The Java class for the between scope.

Universality pattern class

```
public class Universality extends Pattern {

 /* Tells whether the property is violated */
 protected boolean propertyViolated = false;

 /* Tells whether the property might be violated depending on the scope */
 protected boolean propertyPotentiallyViolated = false;

 public boolean checkProperty(Interpreter interpreter) {

  int scopeIsActive = scope.isActive(interpreter);

  /* If the scope is active and the expression is false, the property is violated */
  if (scopeIsActive == Scope.ACTIVE && !checkExpression(interpreter))
   propertyViolated = true;

  /* If the scope is unknown and the expression is false, we have potentially violated the property */
  if (scopeIsActive == Scope.UNKNOWN && !checkExpression(interpreter))
   propertyPotentiallyViolated = true;

  /* If a previously unknown scope becomes true and we had potentially violated
    the property during that time, then we have now definitely violated the property */
  if (scopeIsActive == Scope.POST_ACTIVE) {
   if (propertyPotentiallyViolated)
    propertyViolated = true;
   propertyPotentiallyViolated = false;
  }

  return propertyViolated;
 }
}
```

Figure 23: The Java class for the universality pattern.

variable propertyEncountered is set to true. Otherwise, the scope is checked to see if it was previously active and during that time the condition was not true at least once. In this case, the property is violated, and the value of *propertyViolated* is set to true, indicating that the property has been violated.

Precedence pattern

The Java class for the precedence pattern is shown in Figure 25. This pattern expresses the requirement that while its scope is active, the occurence of one condition (detected by the method *checkExpression2*) requires another condition (detected by the method *checkExpression*) to precede it in occurrence. The *checkProperty* method implements this logic by first determining the status of the scope. If the scope is active or unknown, it is checked whether the second condition has held before the first condition. If so, the property is violated. If the scope was previously active, it is checked whether it was previously determined that the property may have been violated by checking the value of *propertyPotentiallyViolated.* If this is true, then the property has definitely been violated and this is signaled by setting the value of *propertyViolated* to true.

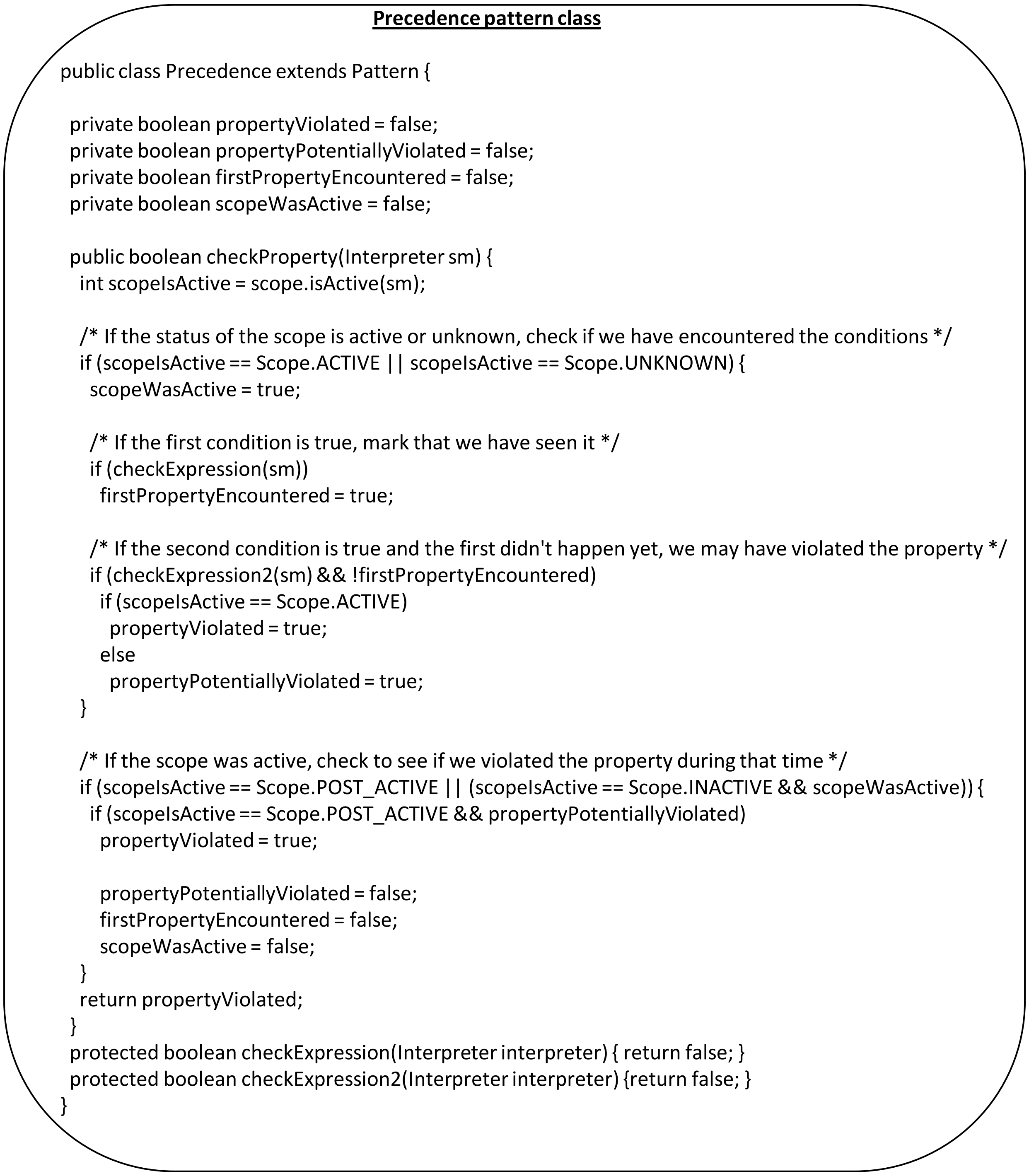
Statechart analysis with Java Pathfinder

The framework analyzes Statechart models using JPF and SPF to perform state exploration. The feature of Statecharts that makes their exploration particularly amenable to symbolic execution by SPF is the combination of a large action language and complex data types. For instance, multi-dimensional arrays can be defined inside a Statechart and used as part of the action language. Reasoning over data types like these can be difficult for some analysis tools, such as those that used a SAT-based [80] or BDD [62] encoding. SPF uses a satisfiability modulo theories (SMT)

Existence pattern class

```
public class Existence extends Pattern {

  private boolean propertyViolated = false;

  private boolean propertyEncountered = false;

  private boolean scopeWasActive = false;

  public boolean checkProperty(Interpreter interpreter) {
    int scopeIsActive = scope.isActive(interpreter);

    /* If the scope is active or unknown and the condition occurs, the property holds*/
    if (scopeIsActive == Scope.ACTIVE || scopeIsActive == Scope.UNKNOWN) {
      scopeWasActive = true;
      if (checkExpression(interpreter))
        propertyEncountered = true;
    }

    /* If the scope occured and the condition didn't occur, the property was violated */
    if (scopeIsActive == Scope.POST_ACTIVE || (scopeIsActive == Scope.INACTIVE && scopeWasActive)) {
      if (!propertyEncountered)
        propertyViolated = true;
      propertyEncountered = false;
      scopeWasActive = false;
    }

    return propertyViolated;
  }
}
```

Figure 24: The Java class for the existence pattern.



Precedence pattern class

```
public class Precedence extends Pattern {

 private boolean propertyViolated = false;
 private boolean propertyPotentiallyViolated = false;
 private boolean firstPropertyEncountered = false;
 private boolean scopeWasActive = false;

 public boolean checkProperty(Interpreter sm) {
  int scopeIsActive = scope.isActive(sm);

  /* If the status of the scope is active or unknown, check if we have encountered the conditions */
  if (scopeIsActive == Scope.ACTIVE || scopeIsActive == Scope.UNKNOWN) {
   scopeWasActive = true;

   /* If the first condition is true, mark that we have seen it */
   if (checkExpression(sm))
    firstPropertyEncountered = true;

   /* If the second condition is true and the first didn't happen yet, we may have violated the property */
   if (checkExpression2(sm) && !firstPropertyEncountered)
    if (scopeIsActive == Scope.ACTIVE)
     propertyViolated = true;
    else
     propertyPotentiallyViolated = true;
  }

  /* If the scope was active, check to see if we violated the property during that time */
  if (scopeIsActive == Scope.POST_ACTIVE || (scopeIsActive == Scope.INACTIVE && scopeWasActive)) {
   if (scopeIsActive == Scope.POST_ACTIVE && propertyPotentiallyViolated)
    propertyViolated = true;

    propertyPotentiallyViolated = false;
    firstPropertyEncountered = false;
    scopeWasActive = false;
  }
  return propertyViolated;
 }
 protected boolean checkExpression(Interpreter interpreter) { return false; }
 protected boolean checkExpression2(Interpreter interpreter) {return false; }
}
```

Figure 25: The Java class for the precedence pattern.

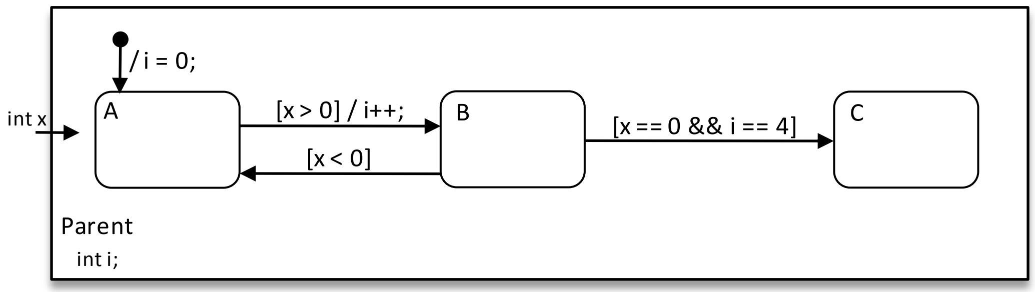


Figure 26: Sample chart for symbolic execution.

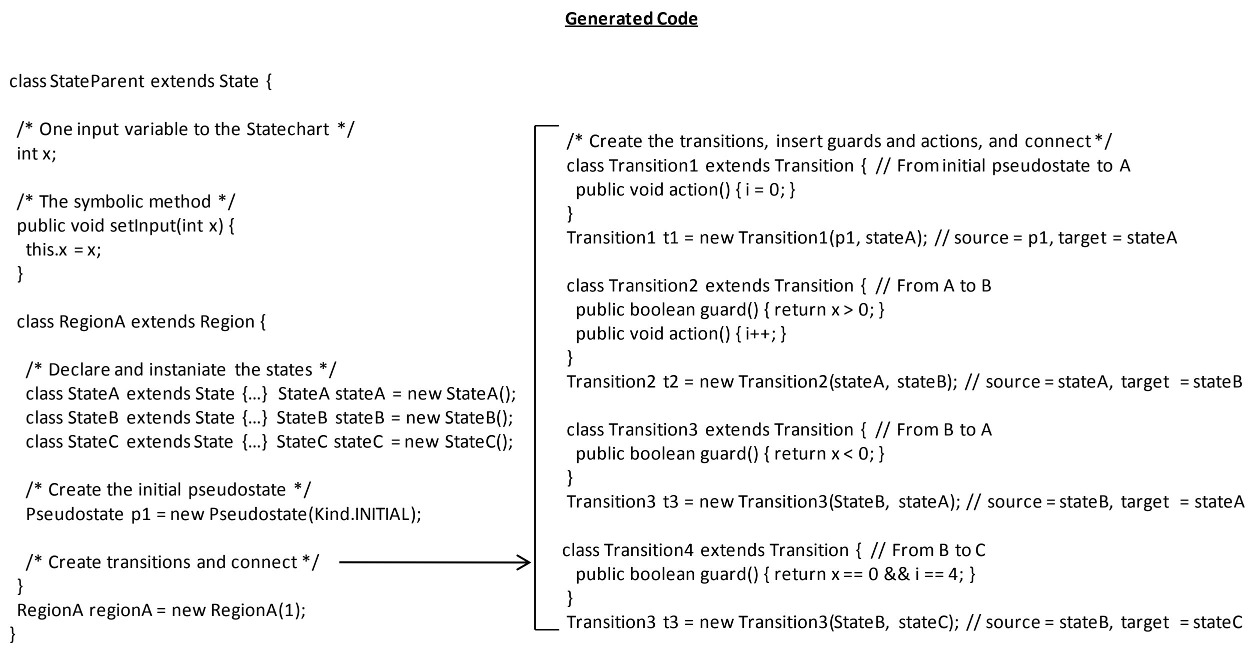
based encoding, which can efficiently reason over data types found in languages like Java and Statecharts.

This section shows how SPF performs symbolic execution over the code to find sequences of inputs that will drive a Statechart through a series of states and how this can be used to perform bounded model checking of Statecharts.

Figure 26 shows a Statechart with three states, A, B and C. Starting from the initial state, A, the goal is to find a sequence of inputs that will cause the Statechart to enter state C. Thus, the goal is to find a sequence of values for the input variable x such that if the model were to use this sequence of values as its inputs, the model would reach state C. The guards on the transitions from A to B and B to C use the input variable x and the internal variable i, while the transition from B to A uses only the input variable x.

The generated structure code for this Statechart is shown in Figure 27. The generated class for the state *Parent* is named *StateParent* and contains an integer variable x that represents the input to the Statechart. The *setInput* method is used to set the value of the input variable, as described in earlier in this chapter.

Symbolic exploration of this model is done the following way. First, a configuration file is created that tells SPF which methods and which parameters to those methods



Generated Code

```
class StateParent extends State {

  /* One input variable to the Statechart */
  int x;

  /* The symbolic method */
  public void setInput(int x) {
    this.x = x;
  }

  class RegionA extends Region {

    /* Declare and instaniate the states */
    class StateA extends State {…} StateA stateA = new StateA();
    class StateB extends State {…} StateB stateB = new StateB();
    class StateC extends State {…} StateC stateC = new StateC();

    /* Create the initial pseudostate */
    Pseudostate p1 = new Pseudostate(Kind.INITIAL);

    /* Create transitions and connect */
  }
  RegionA regionA = new RegionA(1);
}
```

```
/* Create the transitions, insert guards and actions, and connect */
class Transition1 extends Transition { // From initial pseudostate to A
  public void action() { i = 0; }
}
Transition1 t1 = new Transition1(p1, stateA); // source = p1, target = stateA

class Transition2 extends Transition { // From A to B
  public boolean guard() { return x > 0; }
  public void action() { i++; }
}
Transition2 t2 = new Transition2(stateA, stateB); // source = stateA, target = stateB

class Transition3 extends Transition { // From B to A
  public boolean guard() { return x < 0; }
}
Transition3 t3 = new Transition3(StateB, stateA); // source = stateB, target = stateA

class Transition4 extends Transition { // From B to C
  public boolean guard() { return x == 0 && i == 4; }
}
Transition3 t3 = new Transition3(StateB, stateC); // source = stateB, target = stateC
```

Figure 27: Generated structure code for the Statechart in Figure 26.

should be treated symbolically. In this example, the symbolic method is the *setInput* method of the *StateParent* class, and all of its inputs are set to be symbolic. This instructs SPF intercept any concrete invocations of the *setInput* method and replace the parameters with symbolic values. Thus, any use of the symbolic parameters inside the method results in a symbolic value being used instead of a concrete value.

The symbolic values are used during program execution when a branching condition, such as an "if" statement, that depends on a symbolic value is encountered. When such a branching condition is executed, SPF attempts to find two sets of values for all of the symbolic variables used by the branching condition: one set that will cause the "true" branch to be executed, and another set that will cause the "false" branch to be executed. SPF finds these sets of values by invoking an SMT solver, which may attach additional constraints to the symbolic variables.

After creating the configuration file, execution begins. The model is initialized and enters its default state, A. Then, the execution loop of Algorithm 1 begins. This example uses data rather than events to drive the execution, so the first step, "read event," can be ignored. The next step of the execution loop, setting the inputs, results in a call to the symbolic method *setInput* of the *StateParent* class. The value of the parameter to the *setInput* method is assigned to the instance variable x in the *StateParent* class, which results in this instance variable getting a symbolic value.

The next part of the execution loop steps the model, which eventually results in a call to *computeTransitionPath* (Algorithm 3). This algorithm tests the outgoing transitions of current state set, in this case, the outgoing transitions of A. This results in a call to the *guard* method of the *Transition2* class in Figure 27. The transition is enabled if the value of x is greater than 0, which causes the *guard* method to return true. Otherwise, the transition is not enabled. The previous part of the execution loop assigned a symbolic value to x. Because a variable with a symbolic value is used in the conditional statement (*return $x > 0$;*), SPF invokes a constraint solver to find values for x that will cause both branches to be executed. In this case, to explore the true part of the branch and enable the transition, SPF will attach the constraint $x > 0$ to x. To explore the false part of the branch in which the transition is not enabled, SPF will attach the constraint $x <= 0$ to x. By exploring the true part of the branch and enabling the transition, the transition action is executed (*i++*), state A is exited and state B is entered.

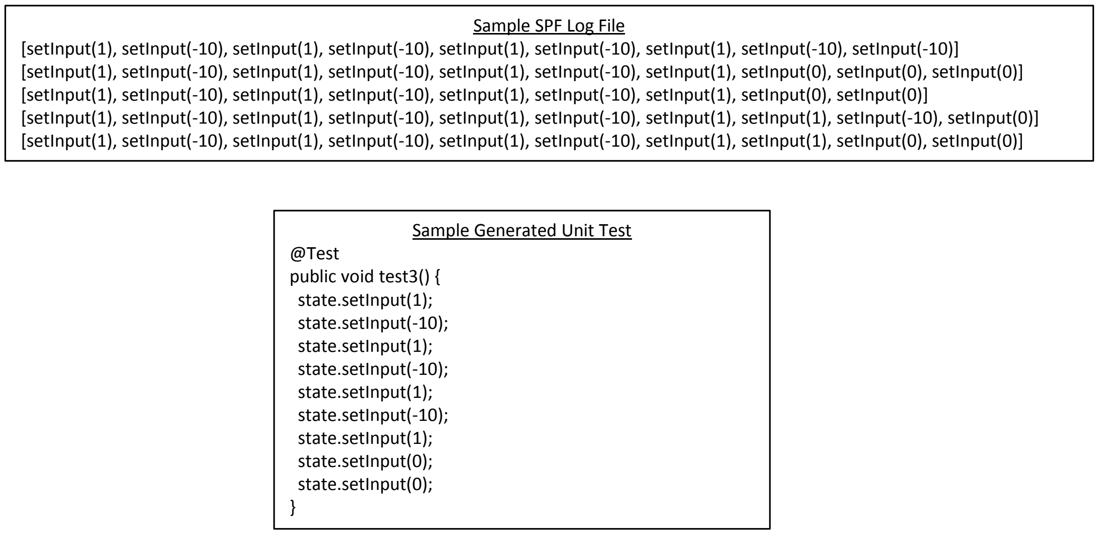
The execution loop then continues with another iteration. Because SPF explored both branches of the conditional described above, there are now two executions maintained by SPF: one in which the current state is A (because the transition was not enabled) and another in which the current state is B (because the transition was enabled). To find inputs that will drive the model to state C, the model must transition from state A to state B four times so that the value of the internal Statechart variable i has the value required to enable the guard from state B to state C. When state B has been entered four times and the execution engine tests the guard on the transition from B to C, SPF tries to find a satisfying assignment to the constraint (*x == 0 && i == 4*). Because i has a concrete value of four, the satisfying assignment (*x = 0*) enables the transition and causes the model to enter state C.

Execution proceeds in this manner until a fixed-point or depth-limit is reached. At the end of execution, SPF reports the input sequences that drive the model through different sequences of states. Two examples of SPF's output are shown in Figure 28. The top of Figure 28 shows sequences of calls to the symbolic method, which in this case is the *setInput* method of the *StateParent* class. The third sequence at the top of Figure 28, for instance, causes the model to go through the states $A, B, A, B, A, B, A, B, C, C$ by providing the inputs $1, -10, 1, -10, 1, -10, 1, 0, 0$. The bottom of the Figure shows the generated unit test corresponding to this test sequence (the instance of the *StateParent* class is named *state*).

These test sequences produced by SPF can be used as inputs to drive the simulation inside the original modeling tools. Alternatively, the generated test sequences can be played back in the Statechart framework because the translation preserves the syntax and hierarchy of the original model, and the state configurations can be seen through the console output.

Case Study

This section describes the application of the Statechart analysis framework to a simplified model of the Mars Exploration Rover (MER). The MER contains a number of different physical devices, such as video cameras and motors, and a number of different software processes, or users, that periodically use the physical devices. To ensure mutual exclusion, the users are not allowed to directly access the physical devices.



Sample SPF Log File

```
[setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(-10)]
[setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(0), setInput(0), setInput(0)]
[setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(0), setInput(0)]
[setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(1), setInput(-10), setInput(0)]
[setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(-10), setInput(1), setInput(1), setInput(0), setInput(0)]
```

Sample Generated Unit Test

```
@Test
public void test3() {
  state.setInput(1);
  state.setInput(-10);
  state.setInput(1);
  state.setInput(-10);
  state.setInput(1);
  state.setInput(-10);
  state.setInput(1);
  state.setInput(0);
  state.setInput(0);
}
```

Figure 28: Sample output log from Symbolic Pathfinder's analysis of the model in Figure 26.

Instead, the users request access to the devices through an arbiter module, which ensures fairness and mutual exclusion.

Figure 29 shows a Simulink model with three Statechart diagrams named *Arbiter*, *User1* and *User2*. The *Arbiter* Statechart models the arbiter module, and the two other Statecharts represent the software processes in the MER arbiter. The model works in the following way. The two users request one of the five resources from the arbiter and wait for a response from the arbiter that tells whether or not access to the resource is granted. Once a resource is granted, the users use it for a period of time and then give control of the resource back to the arbiter. The arbiter reads the resource requests from the users, and, depending on the priority of the requested resources and their current status (available or unavailable), sends a signal to the users that indicates whether the request is granted or denied. Additionally, the arbiter may rescind a user's access to a resource if it receives a request for a conflicting resource with a higher priority from the other user.

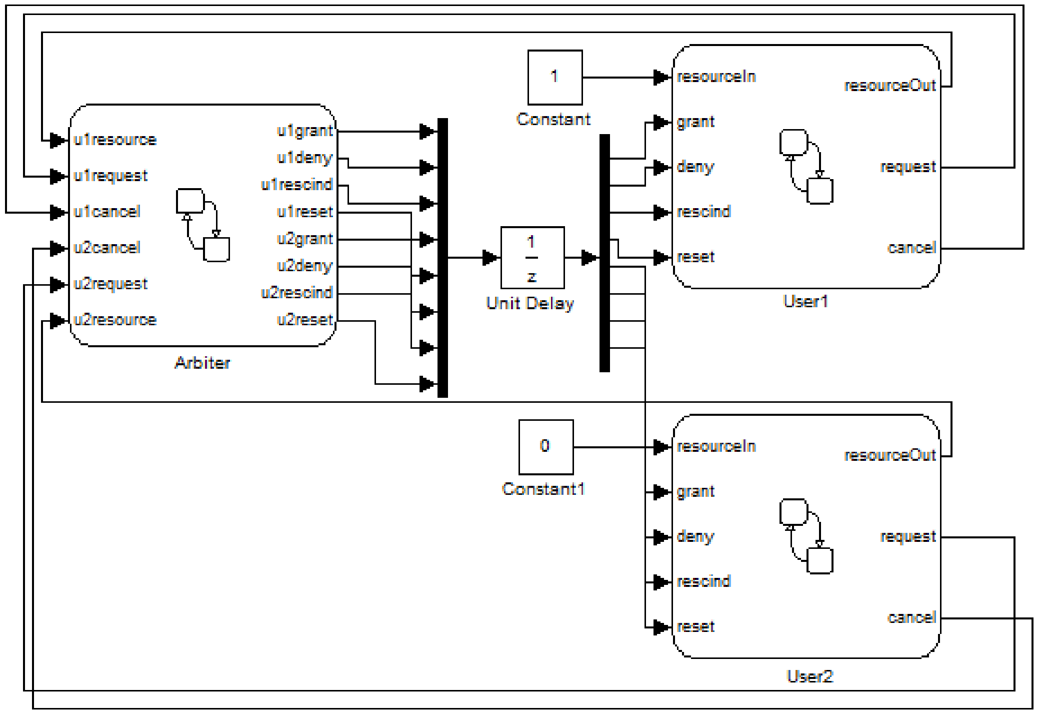


Figure 29: High-level view of the MER Arbiter.

Figure 30 shows the internals of the *User1* Statechart; *User2* is modeled the same way. Initially, the model is in the *Idle* state. A transition to the *Busy* state occurs when one of the five resources is requested (which resource is requested is determined by an external source, shown in Figure 29 as a constant input to *User1*). Upon taking the transition to the *Busy* state, the *resourceOut* variable is set to the value of the requested resource and the *cancel* variable (which indicates whether *User1* is returning control of the resource to the arbiter) is false. These are then passed as inputs to the *Arbiter* module, which decides whether the resource request is granted or denied. The *Arbiter* can also *rescind* a previously granted request if it receives a request for a higher priority, conflicting resource. *User1* also contains a boolean input named *reset*. The intent is that whenever this input is true, it causes *User1* to cancel both its resource and request and transition to the *Idle* state.

As Figure 30 shows, the intent of having the *reset* command take priority over all other inputs is captured by placing the transition guarded by the *reset* command at

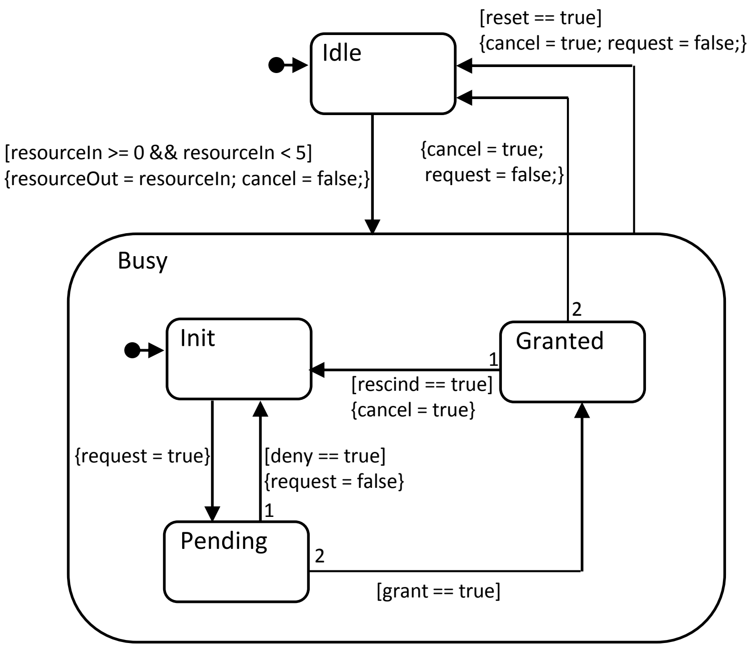


Figure 30: Statechart for User1 from Figure 29.

the highest level of hierarchy (i.e., its source is the *Busy* state). Thus, the property that the transition from state *Pending* to state *Granted* should never be taken when the value of *reset* is true should be satisfied by the model. For a system modeled entirely in Stateflow, the property is satisfied because transitions at higher levels of hierarchy have precedence over transitions at lower levels of hierarchy: when *User1* is in the *Busy* state, if the value of *reset* is true, then the transition from state *Busy* to state *Idle* is taken. However, if either *User1* or *User2* is executed using Rhapsody or UML semantics instead of Stateflow semantics, the system model does not satisfy the property due to the fact that transitions at lower levels of hierarchy are given priority over transitions at a higher level of hierarchy.

The Statechart analysis framework was used to verify whether the property is satisfied depending on the semantics used for *User1*. The first step in applying the framework was to generate Java code representing the structure of the model. The arbiter Statechart contains 33 pseudostates (junctions), 15 atomic states, 2 orthogonal states and 58 transitions, for a total of 108 elements. The generated Java code for the arbiter Statechart is 1788 lines, which corresponds to roughly 16.56 lines of code per element in the Statechart. The Statecharts for the two users each contain 2 pseudostates, 4 atomic states, 1 compound state and 9 transitions, for a total of 16 elements. The generated Java code for each user is 259 lines, yielding approximately 16.19 lines of code per element in the Statechart. The Java translation is linear in the size of the number of Statechart elements due to the fact that it simply the reflects the structure of the Statechart.

The second step in applying the framework was to use SPF to analyze the code and check if there exists a sequence of inputs that will cause the property to be violated. To encode the property that the transition from *Pending* to *Granted* should never be taken if the value of *reset* is true, an assertion stating that the value of *reset* should be false was added to the generated Java code representing the transition action between states *Pending* and *Granted*. SPF searches for input sequences that will cause the assertion to be violated.

Table 1 lists the results of the analysis performed by SPF. Each run is on a different row in the table. The *Arbiter* and *User2* were always executed with Stateflow semantics, while the semantics with which *User1* was executed varied (the semantics used for each experiment are listed in the first column). As the last six rows of Table 1 shows, the property violation was detected by SPF when *User1* was executed with either Rhapsody or UML semantics. The reason that the property can be violated with these semantics is that they both give priority to transitions at lower levels of hierarchy, which is the opposite of the Stateflow semantics.

Table 1: Analysis results for the MER arbiter case study.

Semantics, Seq. size	**Total # Test Cases**	**Property**	**Memory, Time**
User1 Stateflow, 4	125	true	20 M, 43 s
User1 Stateflow, 5	412	true	22 M, 2 m 04 s
User1 Stateflow, 6	1343	true	24 M, 6 m 46 s
User1 UML, 4	57	false	21 MB, 21 s
User1 UML, 5	155	false	21 MB, 53 s
User1 UML, 6	579	false	23 MB, 2 m 50 s
User1 Rhapsody, 4	57	false	21 MB, 21 s
User1 Rhapsody, 5	155	false	21 MB, 55 s
User1 Rhapsody, 6	579	false	23 MB, 2 m 45 s

CHAPTER IV

SEMANTICS WITH FORMULA

This chapter describes an extension to Formula for calculating the execution traces of models. Formula is a modeling language and tool from Microsoft Research that is based on logic-programming, and is used to define, compose and explore DSMLs [83]. The Formula language supports common manipulations of DSMLs such as compositions and extensions, and the analysis tool has a built-in model finding procedure as its formal method.

An execution trace is a sequence of steps that shows how the state of a model evolves during the model's execution. Note the important difference between a model and the *state* of a model. A model combined with a set of rules describing the behavior of the model gives a finite description of a possibly infinite set of *states* that the initial model can reach when it is executed according to the rules that describe its behavior. Execution traces provide a form of simulation and allow one to view a model's reachable states, which can be very helpful for reasoning about non-determinism in languages.

Behavioral semantics in Formula are defined using a set of model transformations. Each transformation takes one input model, produces one output model, and represents one atomic step of execution. Using multiple transformations to define the semantics allows non-concurrent actions to be separated and specified as different atomic steps. A model m is executed using the semantics by giving it to the set of transformations, applying all transformations that take as input a model from the same DSML as m, and then using the output models as the next set of inputs to the transformations. Execution completes when no new outputs are produced.

There were three main problems that had to be addressed when building a tool to compute execution traces. The first is the non-determinism that arises when multiple transformations can be applied to the same input model. For instance, in a modeling language for distributed systems, a transformation T_{Add} might be used to add a node to the system and another transformation T_{Rem} used to remove a node. Using two transformations means that one execution step cannot add *and* remove a node. However, given an input model M_{In}, T_{Add} and T_{Rem} may both be applicable, and the choice of which one to apply is non-deterministic. The trace calculation tool must be able to: (1) determine all transformations that can be applied to an input model at a given step, and (2) create a separate trace for each applicable transformation.

The second problem that was addressed and another source of non-determinism in the semantics is the pattern matching matching inside individual transformations. For instance, the behavioral semantics of Petri nets (formally defined in the next section) state that during a step, one enabled transition may fire. If multiple transitions are enabled during a step, then the choice of which one to fire is non-deterministic. This non-determinism must be made explicit to the trace calculation tool so that a separate trace can be created for each non-deterministic choice. The difficulty lies in doing this in a way that does not rely on any particular DSML. This problem was solved by using a model transformation's parameters to explicitly indicate which parts of a transformation are non-deterministic. The trace calculation tool interprets the parameters to a transformation as ranging over the data instances of the input model to the transformation, which allows a separate trace to be created for each non-deterministic choice made during execution.

The third problem that was addressed deals with storing execution traces after they are computed. The execution traces capture the evolution of a model's state during execution, but from a practical viewpoint, the model itself may also need to be stored along with the trace. Storing a large model at each step of an execution trace can introduce significant storage overhead if a naive approach is used. This problem was solved by designing a component that stores execution traces efficiently by storing only the differences between execution steps whenever possible.

Additionally, a tool for visualizing execution traces was designed. This facilitates an intuitive interpretation of execution traces by allowing users to assign a visual representation to the elements of the DSML and automatically generating a layout of the steps of the trace using these visual representations.

The next section gives examples of execution traces using Petri nets. This is followed by an introduction to the Formula language and a description of the implementation of the trace calculation tool.

Execution traces

An execution trace is a sequence of steps that shows how the state of a model evolves during the model's execution. This section uses Petri nets to show examples of execution traces.

Petri nets

Petri nets [84] are a modeling language especially suited for distributed and concurrent systems because non-determinism can be expressed. They provide a nice combination of simplicity and expressiveness. The formal definition of a basic Petri net is given as a 5-tuple, $PN = (P, T, F, W, M_0)$ where:

- $P = \{p_1, p_2, ..., p_m\}$ is a finite set of *places.*
- $T = \{t_1, t_2, ..., t_m\}$ is a finite set of *transitions.*
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs.
- $W : F \rightarrow \{0, 1, 2, 3, ...\}$ is a weight function.

- $M_0 : P \rightarrow \{0, 1, 2, 3, ...\}$ is the initial marking.

The dynamic behavior is given by the following rules.

- A transition t is enabled if each input place p of t is marked with at least $w(p, t)$ tokens, where $w(p, t)$ is the weight of the arc from p to t.
- An enabled transition may or may not $fire$.
- When an enabled transition t fires, $w(p, t)$ tokens are removed from each input place p of t, and $w(t, p)$ tokens are added to each output place p of t.

Figure 31 shows a graphical representation of a Petri net. Circles are used to represent places, rectangles represent transitions and arrows represent arcs. Small solid dots inside a place represent tokens. Assume that $W(f) = 1 \forall f \epsilon F$, i.e., the weight of all arcs is 1. Given a Petri net with the initial marking on the left side of Figure 31, the transitions labeled *T1* and *T2* are both enabled. The marking that results if *T1* fires is shown on the upper right of Figure 31, while the lower right portion of the Figure shows the resulting marking if *T2* fires. Thus, given the initial marking on the left side of Figure 31, there are two possible execution traces, as shown in Figure 32.

Note that the basic definition of a Petri net given above contains no concept to describe any markings other than the initial marking. Figures 31 and 32 graphically depict the markings after either transition *T1* or *T2* fires using small solid dots to represent the number of tokens in a place. In order for execution traces to be computed, information about the state of a model must be available. The next section shows how a DSML can be extended to include such information.

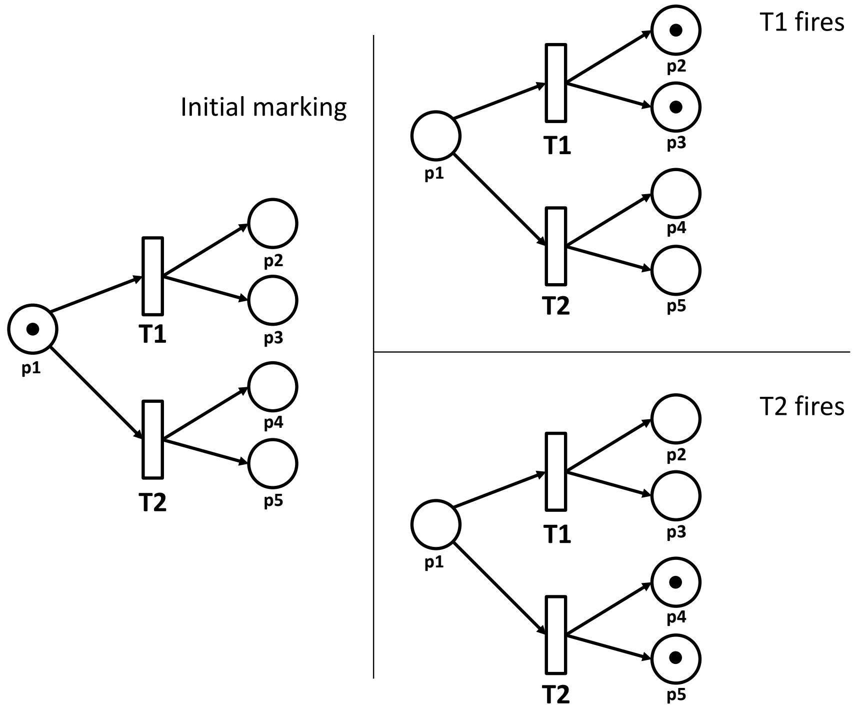


Figure 31: A sample Petri net. Places are represented by circles, transitions by rectangles. A solid dot represents a token. Given the initial marking on the left, transition T1 firing results in the Petri net on the upper right, while the model on the bottom right is the result if transition T2 fires.

Introduction to Formula

Formula, first introduced in [85], is a modeling language and analysis tool for model-based abstractions. A Formula specification consists of a set of data types and declarative rules that define constraints on the data types. Behavioral semantics (next section) are defined using model transformations.

The basic unit of abstraction for defining a DSML is called a *domain*. A domain consists of data types and associated constraints. Figure 33 shows a Formula domain for Petri nets. Line 3 declares a new data type called *Place* for creating places. This declaration can be read like a structure type in C or C++. Each instance of a *Place* has one argument of type Basic named *id*. The Basic type in Formula is a built-in type and includes all of the basic types in Formula: strings, integers and reals. A

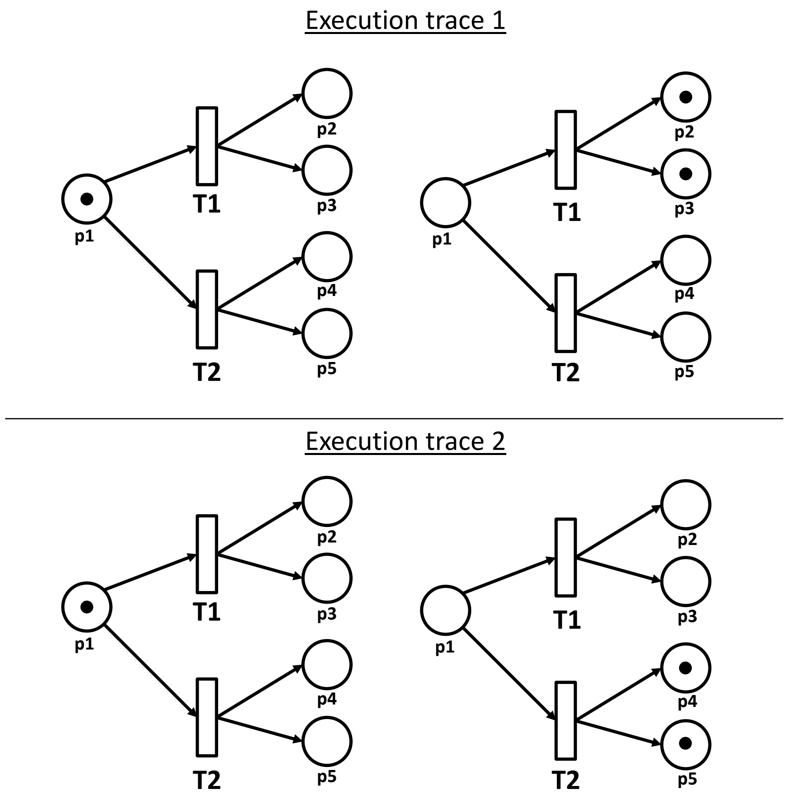


Figure 32: The two execution traces of the Petri net shown in Figure 31.

Place is instantiated by calling the *Place* constructor with an argument of type Basic for its id. For instance, $Place(0)$ and $Place(\text{“}p'')$ instantiate two places with ids of 0 and “p”, respectively. Two instances are equal if and only if they were created by the same constructor applied to the same arguments. Line 4 declares a data type named $Transition$ for representing transitions in a Petri net; $Transition$ also has one argument of type Basic.

Line 6 declares a data type named $TransToPlace$ for representing the arcs between a $Transition$ and a *Place*. The first argument has type $Transition$ and is named *src*, and the second argument has type *Place* and is named *dst*. The third argument has type Real and is named *produces*; this argument represents the number of tokens that are put in the *Place* specified by the second argument if the $Transition$ specified by the first argument fires. The $TransToPlace$ data type also has the *relation* annotation attached to it. This attaches the constraint that both the first and second arguments to this type must be defined. Formula checks that these constraints are satisfied by instances of the domain.

The $PlaceToTrans$ data type (line 8) is used to represent an arc from a *Place* to a $Transition$ and works in an analogous way. The $LoopToPlace$ data type (line 10) is used to represent two arcs: one from its first argument (a *Place*) to its second argument (a $Transition$) that requires a number of tokens given by the third argument (of type Real) in order for the $Transition$ to fire, and a second arc from the second argument (a $Transition$) back to the first argument (a *Place*) that produces the number of tokens given by the fourth argument (of type Real). The $LoopToPlace$ data type is used to reduce the size of models and ease the definition of the semantics; it is syntactic sugar that can be used in place of using both a $PlaceToTrans$ and $TransToPlace$.

The basic *PetriNet* domain in Figure 33 does not contain a way to store the marking of a Petri net. This domain can be extended to include a type to hold in-

```
domain PetriNet
{
  Place          ::=   (id: Basic).
  Transition     ::=   (id: Basic).
  [relation]
 TransToPlace    ::=   (src: Transition, dst: Place, produces: Real).
  [relation]
 PlaceToTrans    ::=   (src: Place, dst: Transition, requires: Real).
  [relation]
  LoopToPlace    ::=   (src: Place, dst: Transition, requires: Real, produces: Real).
}
```

Figure 33: Formula definition of a domain for Petri nets.

```
1.      domain MarkedPetriNet includes PetriNet
2.      {
3.        [function]
4.        Marking    ::=    (place: Place, cnt: Real).
5.      }
```

Figure 34: Extended domain that includes Markings.

formation about a marking, as shown in Figure 34. Line 1 declares a new domain named *MarkedPetriNet* that *includes* the *PetriNet* domain. The “includes” keyword means that the *MarkedPetriNet* domain includes all of the data types and constraints defined in the *PetriNet* domain. The *MarkedPetriNet* domain adds one new data type called *Marking*, which has two arguments: a *Place* and a Real. A *Marking* is used to hold information about the state of a Petri net by indicating that its first argument (a *Place*) contains the number of tokens specified by its second argument (a Real).

A Formula *model* is a finite set of data type instances that satisfy all of the constraints of its associated domain. Figure 35 shows the Formula representation of the Petri net model on the left side of Figure 31. Line 1 declares a model named *m1* whose domain is *MarkedPetriNet*. The notation $p1 is Place(“p1'')$ assigns a place named “p1” to the identifier *p1*. This is syntactic sugar that allows instances to be used across a model without repeatedly constructing them. The five *Place* data type

```
1.     model m1 of MarkedPetriNet
2.     {
3.       p1 is Place("p1")
4.       p2 is Place("p2")
5.       p3 is Place("p3")
6.       p4 is Place("p4")
7.       p5 is Place("p5")
8.       Marking(p1, 1)
9.       Marking(p2, 0)
10.      Marking(p3, 0)
11.      Marking(p4, 0)
12.      Marking(p5, 0)
13.      t1 is Transition("t1")
14.      t2 is Transition("t2")
15.      PlaceToTrans(p1, t1, 1)
16.      PlaceToTrans(p1, t2, 1)
17.      TransToPlace(t1, p2, 1)
18.      TransToPlace(t1, p3, 1)
19.      TransToPlace(t2, p4, 1)
20.      TransToPlace(t2, p5, 1)
21.    }
```

Figure 35: Formula encoding of the Petri net on the left side of Figure 31.

instances defined on lines 3 through 7 are used throughout the rest of the model. Lines 8 through 12 construct five *Marking* data type instances, one for each *Place*. Place *p1* has one token, while the other Places have zero tokens. Lines 13 and 14 create two *Transition* data type instances and assign them to the identifiers *t1* and *t1*, respectively. Lines 15 through 20 create the arcs, all of which have a weight of one.

Behavioral semantics in Formula

Behavioral semantics in Formula are defined as a set of model transformations, each of which takes one model as input and produces one model as output. Additionally, a transformation can have input parameters, which are named and typed, and can be referenced and used in the transformation. Each transformation represents one discrete and atomic step of execution. Inside a transformation are *rules* that describe patterns that should be matched and actions that should be taken when a match is found. All of the computation that takes place inside a single transformation is considered to be atomic, and the output model produced by executing this transformation represents one atomic step of execution applied to the input model.

A model is executed according to the semantics defined by a set of model transformations by providing an input model to the set, applying all compatible transformations from the set, and then using the output models produced by these transformations as the next set of inputs to the set. A transformation T is said to be compatible with a model M if the domain of the input of T is the same as the domain of *M*. Execution completes when a fixpoint is reached: either all of the applicable transformations produce no output or the outputs are identical to the inputs.

At a given point in execution, there may be multiple transformations that are compatible with an input model. Each transformation represents an atomic step of execution, so this means that there are multiple atomic steps of execution that can be performed on the input model. Cast in this light, using multiple transformations to define the semantics provides a way to control the *granularity* of concurrency in the modeling language. Placing a large number of rules in a transformation provides a coarse-grained model of concurrency, while placing relatively few rules in a transformation provides a fine-grained model. In terms of execution traces, when an input model is compatible with multiple transformations at a given step, the trace computing tool must run all of the compatible transformations with the input model and create a separate trace for each.

Figure 36 shows an example of a transformation specified in Formula named *CreateMarkings*. The purpose of this transformation is to create a $Marking$ initialized to 0 for any *Places* in the input model that do not have a corresponding $Marking$. The "from" keyword indicates that the input model belongs to the *MarkedPetriNet* domain and the "to" keyword indicates that the output model belongs to the *Marked-*

PetriNet domain. The “as” keyword on line 1 provides a unique renaming used during the transformation to resolve any naming conflicts between the domains of the input and output models. That is, the “as” keyword allows the elements in the input and output models to be uniquely addressed. The line “MarkedPetriNet as In” prepends an “In” to all of the elements in the input model, thus providing a unique name with which elements in the input model can be referenced.

Transformations consist of a set of *rules*. Each rule consists of a *head* and a *tail*. The tail portion of a rule describes the pattern that must be matched, and the head describes what is created upon each successful match. For instance, the tail of the rule in line 4 of Figure 36 is “In.Place(x)”, which finds instances of the *Place* data type in the input model. The *id* of each *Place* is stored in the variable x. For each such *Place* that is found in the input model, the head of the rule, “Out.Place(x)”, states that a *Place* with the same *id* is created in the output model. Lines 5 through 8 do a similar matching of data instances in the input model and creation of corresponding data instances in the output model.

The interesting rule in the transformation of Figure 36 is on lines 9 through 11. The tail of the rule does three things. First, it searches for *Place* instances in the input model and binds each to the variable p (the rightmost portion of line 9). Line 11 assigns the *id* of this *Place* to the variable x. Line 10 uses the *fail* keyword: the pattern specified by the tail of this rule only matches if there *does not* exist a *Marking* in the input model with the *Place* assigned to the variable p as its first parameter. Intuitively, this rule searches the input model for *Places* for which there is not a *Marking*. For each such match, the head of the rule on line 9 creates a *Marking* for this *Place* in the output model using the *id* of this *Place* and a value of 0 for the number of tokens. Given the input model $m0$ in Figure 37, applying the transformation in Figure 36 results in an output model identical to model $m1$ in Figure 35.

```
transform CreateMarkings                  from MarkedPetriNet as In
                                           to MarkedPetriNet as Out
{
  Out.Place(x)                         :-   In.Place(x).

  Out.Transition(x)                    :-   In.Transition(x).

  Out.TransToPlace(x, y, z)            :-   In.TransToPlace(x, y, z).

  Out.PlaceToTrans(x, y, z)            :-   In.PlaceToTrans(x, y, z).

  Out.LoopToPlace(w, x, y, z)          :-   In.LoopToPlace(w, x, y, z).

  Out.Marking(Out.Place(x), 0)         :-   p is In.Place,

                                             fail In.Marking(p, _),
                                             x = p.id.
}
```

Figure 36: Example transformation in Formula.

```
model m0 of MarkedPetriNet
{
  p1 is Place("p1")
  p2 is Place("p2")
  p3 is Place("p3")
  p4 is Place("p4")
  p5 is Place("p5")
  Marking(p1, 1)
  t1 is Transition("t1")
  t2 is Transition("t2")
  PlaceToTrans(p1, t1, 1)
  PlaceToTrans(p1, t2, 1)
  TransToPlace(t1, p2, 1)
  TransToPlace(t1, p3, 1)
  TransToPlace(t2, p4, 1)
  TransToPlace(t2, p5, 1)
}
```

Figure 37: A model named $m0$, which when given as input to the transformation $CreateMarkings$ in Figure 36 results in the model $m1$ in Figure 35.

Execution traces with Formula

The overall algorithm to compute execution traces is shown in Figure 38. It is shown as part of a class named *TraceCalculator*; the algorithm is presented in an object oriented style for clarity. The *TraceCalculator* class contains four instance variables: a set representing the transformations defining the semantics, a FIFO queue that holds the execution traces, the original input model to the semantics, and an integer representing the maximum number of steps to execute. The method *calculateTraces* begins by pushing a new trace containing only the input model onto the queue of traces. The while loop checks whether the queue of execution traces is empty. If it is not, the top of the queue is popped and the depth limit is checked against this queue. If the depth limit has not been reached, then the last model of the trace is assigned to the variable *inputModel*. Next, all transformations that are compatible with *input-Model* are retrieved; this algorithm is shown in Figure 39 and described in the next section. This is the piece of the trace calculation tool responsible for applying each compatible transformation to the input model. For each compatible transformation, all of the combinations of non-deterministic choices used in that transformation are retrieved from the input model; this algorithm is shown in Figure 40. The transformation is then executed using each combination of non-deterministic choices and using *inputModel* as the input model. If the output of the transformation is non-empty and is not equal to the input, then the output model is added as an extension to the current trace and this new trace is pushed onto the queue of traces. Execution proceeds until the user-specified maximum trace length is reached for all traces or a fixpoint is reached.

Sequencing transformations

The sections above discussed the situations in which multiple transformations can be applied to an input model at a given step. When this situation arises, a separate

```
class TraceCalculator {
 /* The transformations defining the semantics */
 set<Transformation> transforms;

 /* A FIFO queue of traces */
 queue<Trace> traces;

 /* The input model */
 Model input;

 /* The maximum depth to explore */
 int max;

 calculateTraces() {
  // Create new trace with the input model as its only element
  traces.push(new Trace(input));
  while (!traces.empty()) {
   Trace curr = traces.pop();
   if (curr.count < max) {  // check depth limit
    Model inputModel = curr.lastModel;

    /* Apply all applicable transformations to the input model */
    foreach (Transformation t in getCompatibleTransformations(inputModel)) {

     /* Create a new trace for each non-deterministic choice */
     list<Combination<DataInstance>> params = getNondeterministicChoices(t, inputModel);

     /* Run the transformation with each combination of inputs */
     foreach (Combination<DataInstance> currparams in params) {
      t.doTransformation(inputModel, currparams);
      if (t.output != inputModel && t.output != empty) {
       Trace result = curr;    // Create a new trace if the transformation produced output
       result.add(t.output);
       traces.add(result);
      }
     }}}}}}
```

Figure 38: Trace calculation algorithm.

```
set<Transformation> getCompatibleTransformations(Model m) {
  set<Transformation> compatible;
  foreach (Transformation t in transforms) {
    // If the domain of the input model to t is the same
    // as the domain of Model m, add to compatible
    if (t.input.domain == m.domain)
      compatible.add(t);
  }

  return compatible;
}
```

Figure 39: Algorithm to get all valid transformations for an input model.

execution trace needs to be created for each different transformation that can be applied to an input model.

The trace calculation algorithm in Figure 38 does this by iterating over each transformation that is compatible with a given input model. The algorithm to find all of the compatible transformations for a given model is shown in Figure 39. The method is straightforward: for a given model m, iterate over the set of all transformations that define the behavioral semantics. If the domain of the input model to the transformation is the same as the domain of m, the transformation is compatible with m.

Specifying non-determinism in transformations

The second source of non-determinism in the behavioral semantics is the pattern matching inside individual transformations. For example, in the Petri net semantics, multiple transitions can be enabled during a given step, but only one transition is allowed to fire during a single step of execution. In this case, the non-deterministic choice is which enabled transition fires.

The non-deterministic choices must be made explicit to the trace calculating tool so that a separate trace can be created for each such choice. In Formula, this is done by interpreting the *parameters* to a transformation as ranging over the data instances of the input model. When a variable of type T is specified as a parameter to a transformation, the trace calculation algorithm interprets this variable as a non-deterministic choice that ranges over the data instances in the input model. The trace calculation algorithm in Figure 38 calls the *getNondeterministicChoices* method to get all combinations of non-determinism.

Figure 40 shows the *getNonDeterministicChoices* algorithm, which takes two parameters: a Transformation t and a Model m. The algorithm returns all of the combinations of data instances found in the model m that can be used as the parameters to t. The algorithm iterates over each parameter p in the input parameters of t and saves a list of data instances found in m that have the same type as p. Then, all of the possible combinations of these lists of data instances are created and returned.

Storing traces

Once a set of execution traces is computed, they may need to be persisted for later use. Because the models representing the steps of an execution trace may include not only the state of a model but also its structure, significant storage overhead can be introduced if every step of a trace stores both the state (that generally changes between steps) and the structure (which generally remains the same during execution).

To address this issue, a component was built to store execution traces efficiently by storing only the differences between execution steps whenever possible. The main algorithm to perform this storage is shown in Figure 41. The method *storeTrace* takes as input a Trace t and iterates over each model (i.e., step) of the trace. At each step, the algorithm checks if the model of the previous step belongs the same domain as the model of the current step. If so, the *storeDifference* method is used to store only the difference between these two models. First, header information is stored that specifies the current step is being saved as a difference from the previous step. Next, the model elements that were deleted from the previous step are stored,

```
// Algorithm to return all combinations of data instances in the model m that can be used as
// input parameters to the Transformation t
 list<Combination<DataInstance>> getNondeterministicChoices(Transformation t, Model m) {

  /* The parameters to the transformation*/
  set<Parameter> parameters = t.parameters;

  /* For each parameter, store a list of the data instances of that type */
  list<list<DataInstance>> datalist;

  /* List of all of the valid combinations of data instance */
  list<Combination<DataInstance>> combinations;

  foreach (Parameter p in parameters) {
    Datatype dt = p.type;
    list<DataInstance> curr;
      foreach (DataInstance d in m with type dt) {
        curr.add(d);
      }
      datalist.add(curr); // A list of data instances for each parameter
  }

  foreach (list<DataInstance> currlist in datalist) {
    combinations.Add(makeCombo(currlist));
  }
}
```

Figure 40: Algorithm to get the non-deterministic choices in a model.

```
/* Algorithm to store an execution trace */
storeTrace(Trace t) {
  list<Model> sequence = t.models;
  Model prev = none;
  foreach (Model m in sequence) {
    if (m.domain == prev.domain) {
      storeDifference(m, prev);
    }
    else {
      storeModel(m);
    }
    prev = m;
  }
}

/* Stores an entire model */
storeModel(Model m) {
  //Store header

  foreach (DataInstance d in m)
    store(d);
}

/* Stores the differences between the Model m and the Model prev */
storeDifference(Model m, Model prev) {
  // Store header

  /* Record which terms were deleted */
  foreach (DataInstance d in (m – prev))
    storeDeleted(d);

  /* Record which terms were added */
  foreach (DataInstance d in (prev – m)) */
    storeAdded(d);
}
```

Figure 41: Algorithm to store traces.

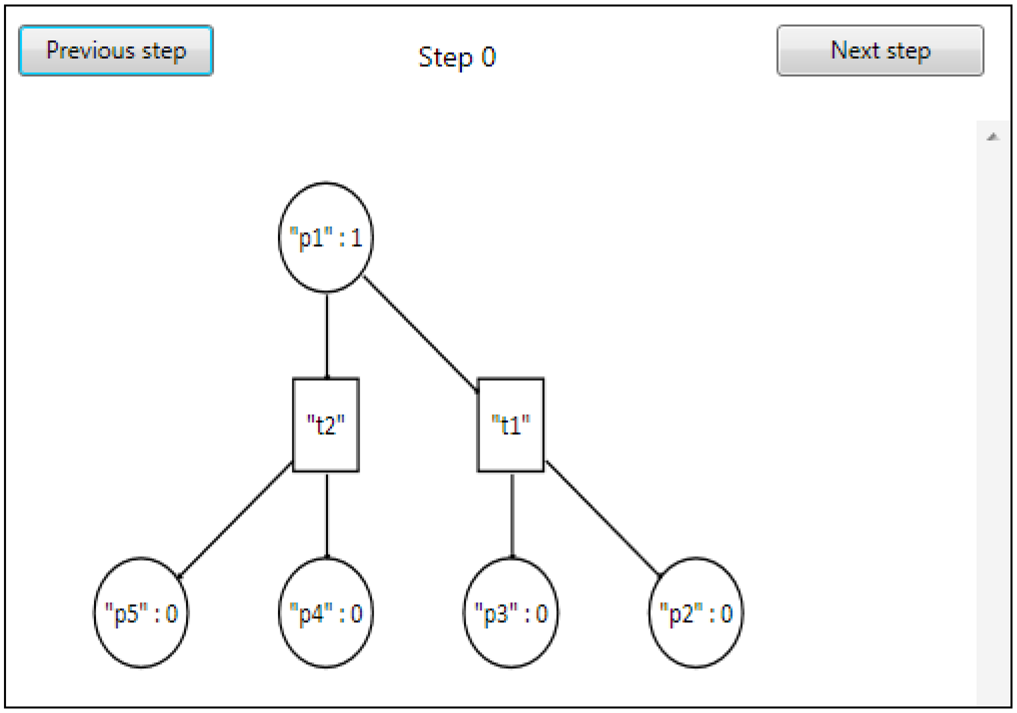
and then the model elements that were added to the previous step are stored. When traces are loaded from storage, steps that are stored as differences are loaded using their previous step and the information describing which data instances were added and deleted.

Visualizing execution traces

To visualize execution traces, a prototype tool was implemented that allows the user to assign a visual element to the concepts in the DSML. These assignments are then used to produce a visual layout of the execution traces. The exact placement of elements is handled by an automatic graph layout library [86].

Figure 42 shows one way the tool can be used to visualize the first execution trace in Figure 32. A label on a *Place* such as "$p1''$: 1 indicates that the *Place* has an *id* of "p1" and contains one token. The top of Figure 42 shows the initial Petri net, and the bottom of Figure 42 shows the Petri net and marking that results if Transition *t1* fires. In the visualization tool, only one step of a trace is shown at a time. The next step in a sequence of traces can be viewed by selecting the "Next step" option in the tool. Figure 43 shows the visualization of the second execution trace in Figure 32.

Initial experiences using the tool to visualize traces have been positive. A wide variety of languages can be intuitively expressed using a small set of visualization primitives, such as circles and squares, along with text labels describing attribute values of elements. This especially includes state-transition type languages, for which the automatic graph layout library [86] was originally designed. Currently, the tool is limited to viewing traces, and there is no mechanism to search for a step of a given trace that satisfies a particular property. This would be a useful addition, as users often want to find the specific step of a trace that satisfies a certain property and then visually examine the previous steps, e.g., for checking how a deadlocked state is reached.



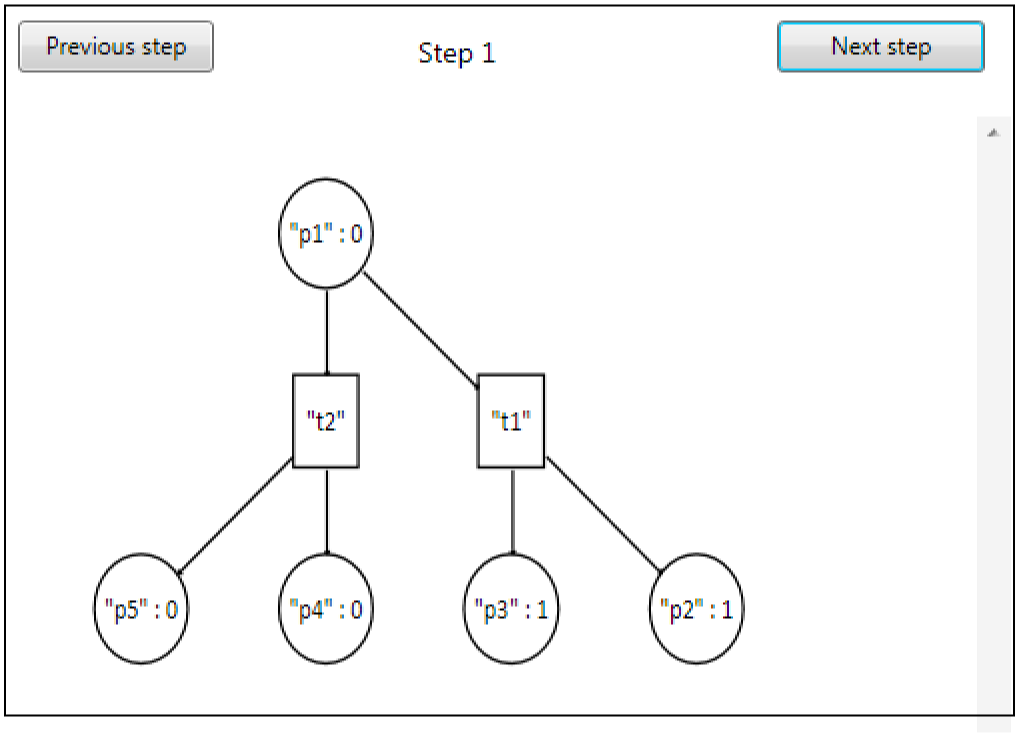


Figure 42: Screenshot of how the trace tool visualizes execution trace 1 in Figure 32.

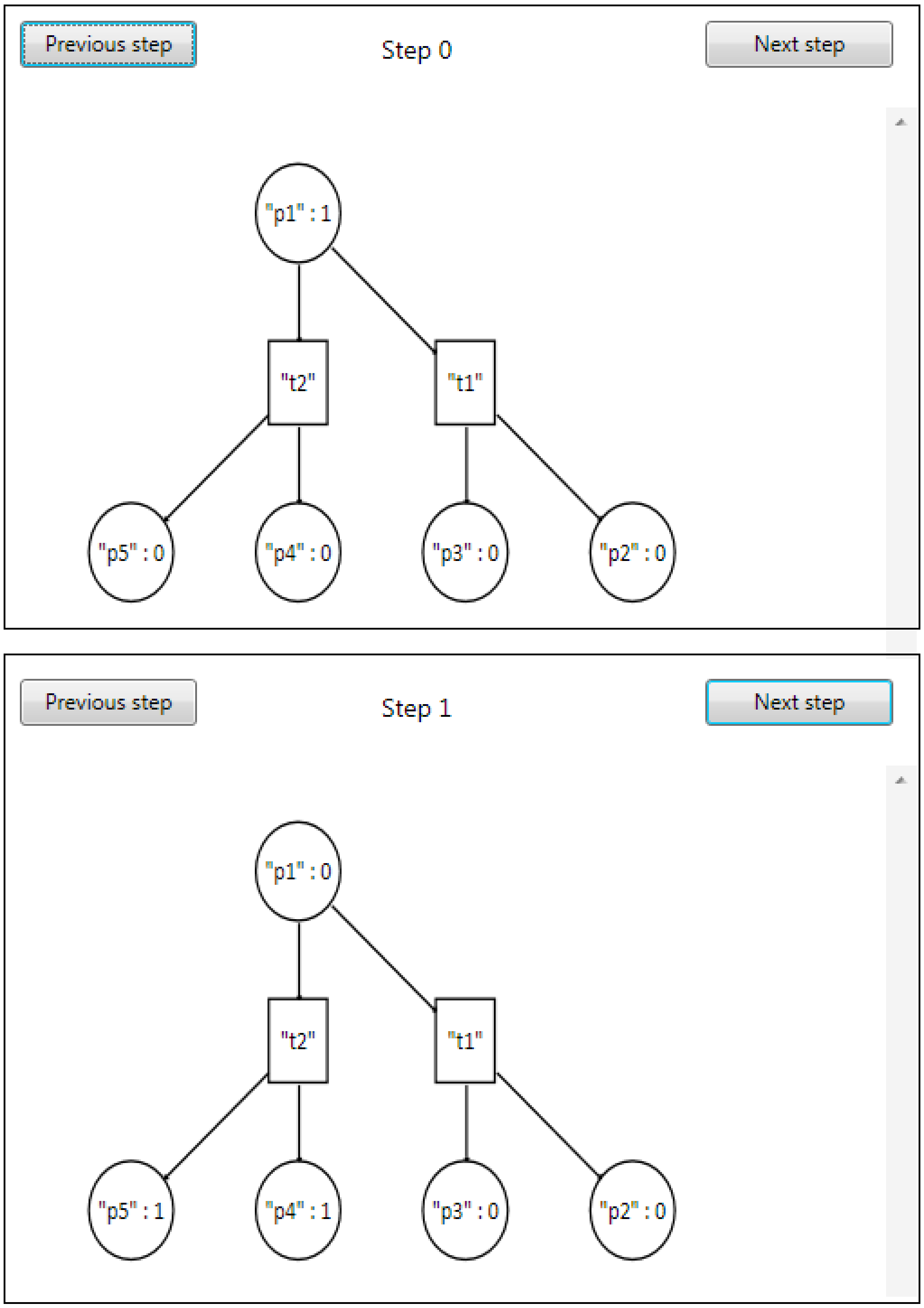


Figure 43: Screenshot of how the trace tool visualizes execution trace 2 in Figure 32.

CHAPTER V

CONCLUSION

This dissertation has described two contributions. The first is a unified framework in which Statechart models of different semantic variants can be defined, simulated and verified. The key idea is that the user describes only the structure of a Statechart model. The structure is then automatically translated into equivalent Java code, and the semantics are selected from a set of pluggable Java components. Components implementing the semantics of three different variants of Statecharts were defined: Matlab/Stateflow, UML and Rhapsody. By decoupling the structure from the semantics, a single model can be executed using multiple semantics, and a system comprised of interacting models using different semantics can be simulated and verified in a single environment. The interaction between models is captured through the input/output interface of the models.

A lightweight method to specify properties that should be monitored was also described. The method is based on the property specification pattern system described in [10]. Properties are specified through an intuitive user interface, from which Java code to monitor these properties is automatically generated. This allows the user to specify a wide-range of commonly occurring properties very quickly.

Analysis and verification is performed using Java Pathfinder, a software model checker, along with Symbolic Pathfinder [15], its symbolic execution engine. The result of the symbolic execution is a set of test-vectors, which represent sequences of inputs that can be given as inputs to a Statechart model to cause its execution to proceed through a certain series of states. Symbolic execution provides a good method of analysis because it allows state exploration to be performed even though system behavior depends on inputs from the environment.

The second contribution described in this dissertation is an extension to Formula, a modeling language and analysis tool, that calculates execution traces of models. The extension consists of three components. The first is a component that applies all applicable transformations to an input model at a given step and creates a separate trace for each such application. The second component is used to create a separate trace for each non-deterministic choice of the input parameters that are passed to a transformation. This makes non-determinism inside a single execution step explicit to the trace computing module. The third component is a tool that stores the execution traces efficiently by computing and storing only the differences between consecutive steps in a trace when possible. Additionally, a prototype tool for visualizing the execution traces was also developed.

Future directions

Both pieces of this work can be extended in several ways. In the Statechart analysis framework, the communication between Statechart models could be modeled in a more complex way to allow analysis under different network conditions. Currently, the Actor model [87] is being investigated as a way of implementing more complex communication. The Actor model of programming uses concurrent, autonomous entities called *actors* which communicate with one another by sending messages.

The method used for analysis in the Statechart framework, symbolic execution, works well for small to medium sized models, but can have difficulty scaling to large models. This can vary based on the types of constraints involving the symbolic inputs. Statechart models that make heavy use of non-linear constraints, for instance, might benefit from a different solver than models that do not contain these kinds of constraints. The biggest source of overhead, though, is the fact that the symbolic execution engine analyzes the code with respect to the semantics of the full Java programming language. Ideally, the symbolic engine would be able to reason about the code using the semantics of the different variants of Statecharts.

An initial port of the Statechart analysis framework from Java to C# has been completed so that the performance of the symbolic execution engine Pex [16] could be evaluated. In some instances, Pex is able to find interesting sequences of inputs on Statechart models for which SPF cannot solve the associated constraints. Further investigation is needed to determine whether this is solely a reflection of the choice of SMT solver being used by each tool.

Generating feedback from the analysis tool that can be easily interpreted on the original model level is a hard problem. Currently, execution traces can either be played back directly in the Statechart framework, or the generated test-sequences can be given as inputs to the original modeling environment and the execution trace can be displayed there. Ideally, the feedback from the analysis tool could be interpreted directly in terms of the original model. The Statechart framework does preserve the syntax and hierarchical structure of a model, but the correspondence between the analysis and the original model can be difficult to see. Ongoing work is attempting to make this correspondence clearer.

For the Formula work, one possible extension is to create a method for defining predicates that are evaluated over the execution traces to provide a way for users to query a set of traces and see which ones satisfy certain properties. The challenge with this and similar problems is defining a specification language that is both powerful and easy to use.

Another extension is to integrate the method of specifying behavioral semantics defined here with Formula's model finding procedure. This would allow users to generate models that are guaranteed to satisfy specified execution conditions in a very efficient way. For instance, the specification could forbid certain states as being reachable, and the symbolic execution would use the definition of the behavioral semantics defined by the transformations to automatically find models satisfying the specification.

APPENDIX A

DIFFERENCES BETWEEN SEMANTIC VARIANTS

Table 2 lists some of the major differences between the variants of Statecharts. In addition to the differences listed in the table, both UML and Rhapsody contain several types of pseudostates, which complicates the calculation of transition paths.

Table 2: Differences between Statechart variants.

	Stateflow	**UML**	**Rhapsody**
Transition hierarchy testing	Top-down	Bottom-up	Bottom-up
Transition action execution	End of path	End of path	Immediately
Condition action execution	Immediately	N/A	N/A
Condition action execution	Immediately	N/A	N/A
Graphical functions	Available	N/A	N/A
Transition priorities	Available	N/A	Available
Parallel state priorities	Available	N/A	Available
Visibility of transition actions	Immediate	End of transition path	Immediate
Internally generated events	Early-return logic	Run to completion	Run to completion

APPENDIX B

BEHAVIORAL SEMANTICS OF PETRI NETS IN FORMULA

```
domain PetriNet
{
  Place         ::=  (id: Basic).
  Transition    ::=  (id: Basic).
  [relation]
 TransToPlace   ::=  (src: Transition, dst: Place, produces: Real).
  [relation]
 PlaceToTrans   ::=  (src: Place, dst: Transition, requires: Real).
  [relation]
  LoopToPlace   ::=  (src: Place, dst: Transition, requires: Real, produces: Real).
}
domain MarkedPetriNet includes PetriNet
{
  [function]
  Marking     ::=  (place: Place, cnt: Real).
}
```

```
transform FireTransition          <firing : In.Transition>      from MarkedPetriNet as In
                                                                 to MarkedPetriNet as Out
 {
   Out.Place(x)                      :-    In.Place(x).
   Out.Transition(x)                 :-    In.Transition(x).
   Out.TransToPlace(x, y, z)         :-    In.TransToPlace(x, y, z).
   Out.PlaceToTrans(x, y, z)         :-    In.PlaceToTrans(x, y, z).
   Out.LoopToPlace(w, x, y, z)       :-    In.LoopToPlace(w, x, y, z).
   disabled(trans)                   :-    trans is In.Transition, trans = firing,
                                            In.PlaceToTrans(x, trans, req),
                                            In.Marking(x, y), req > y.
   disabled(trans)                   :-    trans is In.Transition, trans =
                                            firing, In.LoopToPlace(x, trans, req, _),
                                            In.Marking(x, y), req > y.
   Out.Marking(p2, IncCnt)
   Out.Marking(p2, IncCnt)           :-    trans is In.Transition, trans =
                                            firing, fail disabled(trans),
                                            In.PlaceToTrans(p1, trans, req),
                                            In.Marking(p1, y),
                                            DecCnt = y - req,
                                             In.TransToPlace(trans, p2, prod),
                                            In.Marking(p2, w), IncCnt = w + prod.
   Out.Marking(x, cnt)               :-    trans is In.Transition, trans =
                                            firing, fail disabled(trans),
                                            In.LoopToPlace(x, trans, req, prod),
                                             In.Marking(x, curr),
                                            cnt = curr - req + prod.
   Out.Marking(x, y)                 :-    trans is In.Transition, trans =
                                            firing, disabled(trans),
                                            In.PlaceToTrans(x, trans, _), In.Marking(x, y).
   Out.Marking(x, y)                 :-    trans is In.Transition, trans =
                                            firing, disabled(trans),
                                            In.TransToPlace(trans, x, _), In.Marking(x, y).
   Out.Marking(x, y)                 :-    trans is In.Transition, trans =
                                            firing, disabled(trans),
                                            In.LoopToPlace(x, trans, _, _), In.Marking(x, y).
   Out.Marking(p, z)                 :-    In.Marking(p, z), trans is In.Transition,
                                            trans = firing,
                                            fail In.PlaceToTrans(p, trans, _),
                                            fail In.TransToPlace(trans, p, _),
                                            fail In.LoopToPlace(p, trans, _, _).
     }
```

Figure 44: The behavioral semantics for Petri nets using the MarkedPetriNet domain.

REFERENCES

[1] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "From high-level component-based models to distributed implementations," in *Proceedings of the 10th International conference on Embedded software (EMSOFT)*, 2010, pp. 209–218.

[2] J. E. Rivera and A. Vallecillo, "Adding Behavior to Models," in *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference.* Washington, DC, USA: IEEE Computer Society, 2007, p. 169.

[3] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated Development of Embedded Software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145 – 164, jan. 2003.

[4] "Triskell project (irisa): The metamodeling language kermeta," http://www.kermeta.org, 2006.

[5] S. Cook, G. Jones, S. Kent, and A. Wills, *Domain-specific development with visual studio dsl tools*, 1st ed. Addison-Wesley Professional, 2007.

[6] E. K. Jackson and J. Sztipanovits, "Formalizing the Structural Semantics of Domain-Specific Modeling Languages," *Software and System Modeling*, vol. 8, no. 4, pp. 451–478, 2009.

[7] P. Cousot, "Abstract Interpretation Based Formal Methods and Future Challenges," in *Informatics*, 2001, pp. 138–156.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[9] G. J. Holzmann, "Trends in Software Verification," in *FME*, 2003, pp. 40–50.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*, 1999, pp. 411–420.

[11] A. V. Nori and S. K. Rajamani, "An Empirical Study of Optimizations in YOGI," in *ICSE (1)*, 2010, pp. 355–364.

[12] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[13] E. K. Jackson, W. Schulte, D. Balasubramanian, and G. Karsai, "Reusing Model Transformations While Preserving Properties," in *Fundamental Approaches to Software Engineering (FASE)*, 2010, pp. 44–58.

[14] W. Visser and P. C. Mehlitz, "Model checking programs with java pathfinder," in *Model Checking Software, 12th International SPIN Workshop*, 2005, p. 27.

[15] S. Anand, C. S. Pasareanu, and W. Visser, "JPF-SE: A Symbolic Execution Extension to Java PathFinder," in *TACAS*, 2007, pp. 134–138.

[16] N. Tillmann and J. de Halleux, "Pex-White Box Test Generation for .NET," in *TAP*, 2008, pp. 134–153.

[17] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston, *Object-oriented Analysis and Design with Applications, Third Edition.* Addison-Wesley Professional, 2007.

[18] E. M. Clarke, "The Birth of Model Checking: History, Achievements, Perspectives," in *25 Years of Model Checking*, 2008, pp. 1–26.

[19] H. Grönniger, J. O. Ringert, and B. Rumpe, "System Model-Based Definition of Modeling Language Semantics," in *FMOODS '09/FORTE '09: Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems.* Berlin, Heidelberg: Springer-Verlag, 2009, pp. 152–166.

[20] Object Management Group, *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/2006-01-01

[21] ——, "Unified Modeling Language (UML), Superstructure, V2.1.2," Tech. Rep., November 2007. [Online]. Available: http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF

[22] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[23] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[24] D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of "Semantics"?" *Computer*, vol. 37, pp. 64–72, 2004.

[25] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson, "Semantic Anchoring with Model Transformations," in *In ECMDA-FA, volume 3748 of LNCS.* Springer, 2005, pp. 115–129.

[26] B. W. Kernighan and D. Ritchie, *The C Programming Language, Second Edition.* Prentice-Hall, 1988.

[27] R. D. Tennent, "The Denotational Semantics of Programming Languages," *Commun. ACM*, vol. 19, no. 8, pp. 437–453, 1976.

[28] G. D. Plotkin, "A Structural Approach to Operational Semantics," University of Aarhus, Tech. Rep. DAIMI FN-19, 1981. [Online]. Available: http://citeseer.ist.psu.edu/plotkin81structural.html

[29] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *COMMUNICATIONS OF THE ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[30] L. Allison, *A Practical Introduction to Denotational Semantics.* New York, NY, USA: Cambridge University Press, 1986.

[31] H. P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics (Studies in Logic and the Foundations of Mathematics, Volume 103). Revised Edition*, revised ed. North Holland, November 1985.

[32] P. D. Mosses, "Compiler Generation Using Denotational Semantics," in *MFCS*, 1976, pp. 436–441.

[33] J. Bodwin, L. Bradley, K. Kanda, D. Litle, and U. F. Pleban, "Experience with an Experimental Compiler Generator Based on Denotational Semantics," in *SIGPLAN Symposium on Compiler Construction*, 1982, pp. 216–229.

[34] P. D. Mosses, "The Varieties of Programming Language Semantics," in *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI*, 2001, pp. 165–190.

[35] E. Börger and R. F. Stärk, *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.

[36] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing, "An Event-Based Structural Operational Semantics of Multi-Threaded Java," in *Formal Syntax and Semantics of Java, volume 1523 of Lecture Notes in Computer Science.* Springer, 1998, pp. 157–200.

[37] M. W. Whalen, "A Parametric Structural Operational Semantics for Stateflow, UML Statecharts, and Rhapsody," University of Minnesota Software Engineering Center, 200 Union St., Minneapolis, MN 55455, Tech. Rep. 2010-1, August 2010.

[38] J. Palsberg and M. I. Schwartzbach, "Safety Analysis versus Type Inference," *INFORMATION AND COMPUTATION*, vol. 118, pp. 128–141, 1995.

[39] C. Calcagno, S. Helsen, and P. Thiemann, "Syntactic Type Soundness Results for the Region Calculus," *Information and Computation*, vol. 173, pp. 173–2.

[40] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods.* Oxford University Press, 1993, pp. 9–36.

[41] D. Varró, "Automated Formal Verification of Visual Modeling Languages by Model Checking," *Software and System Modeling*, vol. 3, no. 2, pp. 85–113, 2004.

[42] R. W. Floyd, "Assigning Meanings to Programs," *Mathematical aspects of computer science*, vol. 19, no. 19-32, p. 1, 1967.

[43] E. W. Dijkstra, "Go to Statement Considered Harmful," pp. 351–355, 2002.

[44] J. W. d. Bakker, *Mathematical Theory of Program Correctness.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1980.

[45] T. Massoni, R. Gheyi, and P. Borba, “A UML Class Diagram Analyzer,” in *In 3rd International Workshop on Critical Systems Development with UML, affiliated with 7th UML Conference*, 2004, pp. 143–153.

[46] D. Jackson, “A Comparison of Object Modelling Notations: Alloy, UML and Z,” Tech. Rep., 1999.

[47] V. D’Silva, D. Kroening, and G. Weissenbacher, “A Survey of Automated Techniques for Formal Software Verification,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 7, pp. 1165–1178, June 2008. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2008.923410

[48] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems.* New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[49] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *POPL*, 1977, pp. 238–252.

[50] G. S. Boolos, J. P. Burgess, and R. C. Jeffrey, *Computability and Logic, Fifth Edition.* Cambridge University Press, 2007.

[51] P. Lacan, A. Deutsch, G. Gonthier, J. N. Monfort, and L. V. Q. Ribal, “ARIANE 5 - The Software Reliability Verification Process,” *Proceedings DASIA 98 - Data Systems in Aerospace.*

[52] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The Software Model Checker BLAST: Applications to Software Engineering,” *INT. J. SOFTW. TOOLS TECHNOL. TRANSFER*, 2007.

[53] F. Logozzo, “Language-agnostic Contract Specification and Checking with Code-Contracts and Clousot,” in *Bytecode*, 2010.

[54] “The Clang Static Analyzer,” http://clang-analyzer.llvm.org/.

[55] E. M. Clarke and E. A. Emerson, “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic,” in *Logic of Programs*, 1981, pp. 52–71.

[56] J.-P. Queille and J. Sifakis, “Specification and verification of concurrent systems in CESAR,” in *Symposium on Programming*, 1982, pp. 337–351.

[57] E. M. C. Jr., O. Grumberg, and D. A. Peled, *Model Checking.* The MIT Press, 1999.

[58] Y.-A. Chen, E. M. Clarke, P.-H. Ho, Y. V. Hoskote, T. Kam, M. Khaira, J. W. O'Leary, and X. Zhao, "Verification of All Circuits in a Floating-Point Unit Using Word-Level Model Checking," in *Formal Methods in Computer-Aided Design (FMCAD)*, 1996, pp. 19–33.

[59] G. J. Holzmann, "Software Model Checking with SPIN," *Advances in Computers*, vol. 65, pp. 78–109, 2005.

[60] P. Godefroid, "Model checking for Programming Languages Using VeriSoft," in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York, NY, USA: ACM, 1997, pp. 174–186.

[61] S. Demri, F. Laroussinie, and P. Schnoebelen, "A Parametric Analysis of the State-Explosion Problem in Model Checking," *J. Comput. Syst. Sci.*, vol. 72, no. 4, pp. 547–575, 2006.

[62] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic Model Checking: $10\hat{2}0$ States and Beyond," in *Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990, pp. 428–439.

[63] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[64] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. Lecture Notes in Computer Science. Springer, 1996, vol. 1032.

[65] G. J. Holzmann, *Design and Validation of Computer Protocols.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[66] M. Y. Vardi, "Automatic Verification of Probabilistic Concurrent Finite-State Programs," in *Foundations of Computer Science (FOCS)*, 1985, pp. 327–338.

[67] P. Zuliani, A. Platzer, and E. M. Clarke, "Bayesian Statistical Model Checking with Application to Simulink/Stateflow Verification," in *Hybrid Systems: Computation and Control (HSCC)*, 2010, pp. 243–252.

[68] S. Anand, C. S. Pasareanu, and W. Visser, "JPF-SE: A Symbolic Execution Extension to Java PathFinder," in *TACAS*, 2007, pp. 134–138.

[69] Robby, M. B. Dwyer, and J. Hatcliff, "Domain-specific Model Checking Using The Bogor Framework," *Automated Software Engineering, International Conference on*, vol. 0, pp. 369–370, 2006.

[70] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 267–280.

[71] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A Theorem Prover for Program Checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[72] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[73] K. R. M. Leino and P. Rümmer, "A Polymorphic Intermediate Verification Language: Design and Logical Encoding," in *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS*, 2010, pp. 312–327.

[74] L. M. de Moura and N. Bjørner, "Satisfiability Modulo Theories: An Appetizer," in *SBMF*, 2009, pp. 23–36.

[75] M. Moskal, "Satisfiability Modulo Software," Ph.D. dissertation, University of Wroclaw, Poland, 2009.

[76] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Trans. Softw. Eng.*, vol. 2, no. 3, pp. 215–222, 1976.

[77] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[78] M. von der Beeck, "A Comparison of Statecharts Variants," in *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, 1994, pp. 128–148.

[79] J. Niu, J. M. Atlee, and N. A. Day, "Template semantics for model-based notations," *IEEE Trans. Software Eng.*, vol. 29, no. 10, pp. 866–882, 2003.

[80] N. Eén and N. Sörensson, "An Extensible SAT-Solver," in *Theory and Applications of Satisfiability Testing*, 2004, pp. 333–336. [Online]. Available: http://www.springerlink.com/content/x9uavq4vpvqntt23

[81] J. Porter, G. Hemingway, H. Nine, C. vanBuskirk, N. Kottenstette, G. Karsai, and J. Sztipanovits, "The esmol language and tools for high-confidence distributed control systems design. part 1: Language, framework, and analysis," September 2010.

[82] D. Harel and H. Kugler, "The rhapsody semantics of statecharts (or, on the executable core of the uml) - preliminary version," in *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, 2004, pp. 325–354.

[83] E. K. Jackson, D. Seifert, M. Dahlweid, T. Santen, N. Bjørner, and W. Schulte, "Specifying and composing non-functional requirements in model-based development," in *Software Composition*, 2009, pp. 72–89.

[84] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541 –580, apr. 1989.

[85] E. K. Jackson and J. Sztipanovits, "Towards a formal foundation for domain specific modeling languages," in *EMSOFT*, 2006, pp. 53–62.

[86] L. Nachmanson and L. Powers, "Microsoft automatic graph layout library (msagl)," http://research.microsoft.com/en-us/projects/msagl/, 2008.

[87] G. Agha, *Actors: a model of concurrent computation in distributed systems.* Cambridge, MA, USA: MIT Press, 1986.