

Polyglot: Modeling and Analysis for Multiple Statechart Formalisms

Daniel Balasubramanian
Vanderbilt University/ISIS
1025 16th Ave S
Nashville, TN 37212
daniel@isis.vanderbilt.edu

Corina S. Pășăreanu
Carnegie Mellon/NASA Ames
Research Center
M/S 269-2
Moffett Field, CA 94035
corina.s.pasareanu@nasa.gov

Michael W. Whalen
Dept. of Computer Science
and Engineering
University of Minnesota
Minneapolis, MN 55455
whalen@cs.umn.edu

Gabor Karsai
Vanderbilt University/ISIS
1025 16th Ave S
Nashville, TN 37212
gabor@isis.vanderbilt.edu

Michael Lowry
NASA Ames Research Center
M/S 269-2
Moffett Field, CA 94035
Michael.R.Lowry@nasa.gov

ABSTRACT

In large programs such as NASA Exploration, multiple systems that interact via safety-critical protocols are already designed with different Statechart variants. To verify these safety-critical systems, a unified framework is needed based on a formal semantics that captures the variants of Statecharts. We describe Polyglot, a unified framework for the analysis of models described using multiple Statechart formalisms. In this framework, Statechart models are translated into Java and analyzed using pluggable semantics for different variants operating in a polymorphic execution environment. The framework has been built on the basis of a parametric formal semantics that captures the common core of Statecharts with extensions for different variants, and addresses previous limitations. Polyglot has been integrated with the Java Pathfinder verification tool-set, providing analysis and test-case generation capabilities. We describe the application of this unified framework to the analysis of NASA/JPL's MER Arbiter whose interacting components were modeled using multiple Statechart formalisms.

Categories and Subject Descriptors

H.4 [Model-Based Design]: Analysis, Testing

General Terms

DESIGN, LANGUAGES, VERIFICATION

Keywords

Statecharts, Semantics of Models, Polymorphism

1. INTRODUCTION

In many real-world domains, software systems are integrated from components built in multiple development environments and need to be verified against system and lower-level requirements. Model-driven development is increasingly used in the design and implementation of safety and mission-critical systems. For example, NASA's human space program is transitioning to model-based software development, with software subsystems designed, and often source code directly generated from different UML models and other modeling formalisms such as Matlab.

The transition to model-based development is motivated both by lower costs for overall software development and by the enhanced ability to find defects early in the design cycle. Models provide better understanding than source code for engineers in many disciplines, and are amenable to analysis [5] that provides confidence and guarantees about system behavior. Verification and validation techniques exist for several individual modeling formalisms, and supporting tools offer features such as test-input generation and model checking. However, existing modeling languages and analysis tools target only a single formalism and have limited use for analyzing models from multiple formalisms.

This paper presents our work on Polyglot, an extensible framework with supporting tools for the design-time analysis of model-based flight and ground control software that is developed with multiple modeling formalisms.

Polyglot provides a unified environment in which multiple variants of Statecharts [14], a popular modeling formalism for the dynamics of reactive systems, can be executed and their models verified against properties. This approach provides several useful benefits. First, it allows a user to understand and analyze the behavior of models across different tools in a single framework. Second, this approach allows users to verify whether model properties are preserved across different variants of Statecharts. A model can be created in one tool and then simulated using different semantics, ensuring that there are no misunderstandings in requirements and design development due to semantical differences. Third, our unified environment provides the basis for analyzing interacting models that operate under different semantics. This is crucial to finding interoperability and in-

terface errors early in the design phase, since previous analyses have found that the majority of errors in NASA’s Apollo and Skylab software were interface errors [11].

We perform the analysis of the models by translating them to a common representation. The key is that we translate the structure of the models to the common representation and define the behavior, thus the semantics, as a “pluggable” component: an execution engine for the common representation. This allows each model to be simulated and analyzed with multiple semantic variants. To provide confidence that our semantics are faithful to those of the original tools, we have also developed a formal description of the Statecharts semantics written in the structural operational semantics formalism (SOS) [26]. We note that discovering the formal semantics of proprietary modeling tools has been a daunting task, since the available documentation is usually informal, often incomplete and sometimes ambiguous. Completing our SOS formal descriptions required considerable experimentation to discover the real semantics. We used the formal description in two ways: to ensure that our framework correctly describes the behavior of the Statechart variants and to understand the corner cases of each notation. The analysis of the translated models is performed using Java Pathfinder [1]: a verification tool-set that incorporates software model checking and test-case generation capabilities, based on symbolic execution techniques [22].

This paper makes the following contributions. We provide a unified framework for modeling and analysis using multiple Statecharts formalisms. We describe a generic translation from the Statecharts modeling formalisms into a common Java representation. This translation captures the structure of a Statechart model, but omits the translation of the behavior. The behavior is defined in separate Java modules, which allows a model to be simulated and analyzed with different semantic variants. The modules implementing the various semantics were developed and tested in accordance with a formal description, providing confidence that we are faithful to the original tools. We give a formal description of the semantics of Statecharts across several variants, namely UML, Rhapsody, and Matlab’s Stateflow. This description corrects several limitations present in previous attempts at formalization. Finally we provide an implementation for our framework and a case study – the modeling and analysis of NASA/JPL’s MER Arbiter, whose interacting components have been modeled using different Statechart formalisms.

The rest of the paper is structured as follows. Section 2 provides a brief background on Statecharts and on the Java Pathfinder tool-set. Section 3 describes the Polyglot framework and Section 4 describes our formalization of Statechart semantics. We compare with related work in Section 6 and conclude in Section 7.

2. BACKGROUND

The modeling formalism we target is Statecharts, a graphical modeling language that allows the hierarchical and parallel composition of finite state machines. As described in detail in Section 6, there is an abundance of tools for defining Statecharts, each of which has a distinct semantics. We focus on three in particular due to their popularity: Rhapsody [15] from Rational/IBM, Simulink/Stateflow [19] from the Mathworks and UML State Machine semantics [10].

2.1 Statecharts

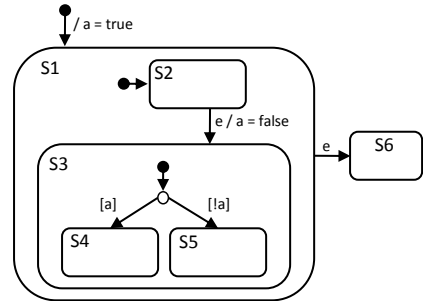


Figure 1: Example Statechart.

The main entity of a Statechart is a *state*, which can contain hierarchically nested states, also called sub-states. A state that contains other states can have either a *sequential* or *orthogonal* decomposition. When a state has a sequential decomposition, it means that when it is active, exactly one of its sub-states is active. An orthogonal decomposition means that when a state is active, all of its sub-states are also active.

A *transition* connects its source state to its target state. A transition may have an associated *trigger event*, *guard* and various *actions*. Events are signals that can be *present* or *absent* and may have an associated *value*. The guard is a predicate evaluated over the chart’s data and current set of active states. A transition can be executed when an event in its trigger is present and its guard evaluates to true. The associated transition actions are also performed when the transition is taken.

Consider the Statechart shown in Figure 1, which is derived from the example in [7]. This seemingly simple chart is enough to demonstrate some of the differences in Statechart semantics. After the chart executes its initial transitions, the chart will be in state $S2$ and the value of a is set to *true*. Upon the occurrence of event e , each semantic variant will end in a different state. The Stateflow semantics will terminate in state $S6$ because precedence is given to the outermost enabled transition. The UML State Machine semantics will terminate in state $S4$ because transition actions (which in this case set the value of a to *false*) are not evaluated until the end of a reaction. The Rhapsody semantics perform the transition actions as they are encountered and will terminate in state $S5$.

2.2 Verification and test case generation

Java Pathfinder (JPF) is an open-source tool-set for verifying Java bytecode. It includes an explicit-state model-checker (`jpf-core`) and several extensions such as Symbolic PathFinder (`jpf-symbc`) [22] that we use in our work. The model checker consists of an extensible custom Java Virtual Machine (JVM), listener support for monitoring and influencing JPF’s search, and a set of Java methods for instrumenting Java programs, e.g. to introduce non-deterministic choices in the execution of the program under test via Choice-Generators. JPF’s default mode of execution, termed concrete execution, performs explicit-state model checking over Java bytecode.

Symbolic Pathfinder (SPF) is an extension to JPF that performs symbolic execution for generating test cases that achieve high test coverage. Symbolic execution [17] is a well-

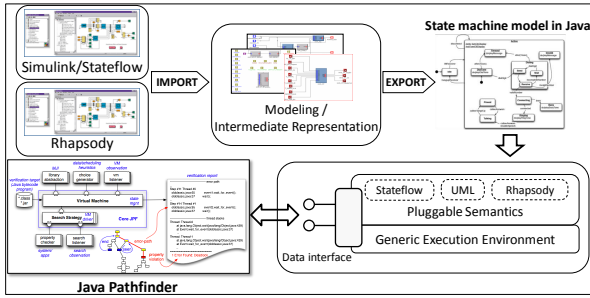


Figure 2: The Polyglot framework.

known program analysis that uses symbolic values instead of actual data as inputs and symbolic expressions to represent the values of program variables. The state of a symbolically executed program includes the symbolic values of program variables, a path condition (PC), and a program counter. The path condition is a Boolean formula over the symbolic inputs, encoding the constraints which the inputs must satisfy in order for an execution to follow the particular associated path. These conditions are solved using off-the-shelf constraint solvers to generate test cases (test input and expected output pairs) guaranteed to exercise the analyzed code. Symbolic PathFinder generates both test vectors and test sequences; the latter are necessary for testing looping, reactive programs.

3. THE POLYGLOT FRAMEWORK

Figure 2 illustrates our framework. Polyglot is used with a methodology that consists of the following steps: (1) translate from the modeling tools into an intermediate representation (IR), (2) translate from the IR to Java code that represents the state structure of a hierarchical, parallel finite state machine, (3) select the desired semantics from a set of pluggable semantics implemented in Java, (4) combine the structure code with the pluggable semantics code to execute the chart, and (5) analyze using Java Pathfinder.

We chose Java as the common language to represent and analyze Statecharts for several reasons. First, we needed an *executable* representation for the models, to allow for quick validation and debugging. We also wanted a *modular* and *extensible* design for our framework, to allow for easy integration of new semantic variants. Java is an ideal language for this purpose. Furthermore, we chose Java in order to leverage the model checking and symbolic execution techniques from the Java PathFinder tool-set (SPF). This allows us to use a high-level language to express the semantics and leverage existing verification and testing technology to do the analysis.

In addition, the Statechart variants that we have studied have large action languages. Features like complex data types and function states, along with transitions containing guards and actions that use these types and functions, are difficult to represent in simpler modeling languages (e.g. satisfiability modulo theories (SMT) formulas that can be solved with off-the-shelf solvers). On the other hand, there is a straightforward mapping from most features of the action languages into a corresponding or similar concept in Java. As it happens, we do use SMT technology in our symbolic execution techniques, but instead of encoding the

semantics directly in terms of SMT formulas, we encode the semantics in Java and let the SPF tool generate and solve the symbolic formulas. This is much easier than defining an SMT encoding of the semantics and properties manually.

The first step in our methodology translates from individual modeling tools into the IR. We have built translators for IBM Rational Rhapsody [15], Simulink / Stateflow [19] and UML variants. In addition to translating the syntactic description of the models into IR, we use the extension features of each tool to allow the user to insert custom annotations that are also passed to the IR and inserted into the generated Java code. From the IR, Java code that represents the structure of the model is generated: when the code is executed it builds an object structure similar to the structure of the model. This “structure” code is combined with the semantic modules to provide the execution of the model.

We have designed the generated code and semantic modules so that they work together to provide a clean input-output interface to the environment. This interface allows us to simulate the models and also to connect them to JPF, with JPF controlling the execution of the model non-deterministically. It also allows us to perform *symbolic execution* of our models using SPF. This means that in addition to property checking provided by SPF, we are also able to perform test-vector generation for multiple semantic versions of the same model.

3.1 Translation to Java

The first step in our framework translates models from individual tools into an intermediate representation. The IR is constructed in the context of the Generic Modeling Environment (GME) [18], a tool for building and editing domain-specific modeling languages (DSMLs). In GME, a *meta-model* describes the domain concepts and their relationships. For example, our meta-model includes concepts such as States, Transitions and Events, and relationships such as States containing States to model the hierarchy that can be found inside Statecharts. In addition to Statechart specific features, our meta-model also defines concepts for analysis that are used later in the toolchain. For instance, annotations describing State attributes, like ‘error state’, are added to the intermediate GME models, and are later translated into specific annotations for the model checker.

From the intermediate representation (which also has an XML representation) we generate Java code that represents the structure of a Statechart. The fact that we generate only the structure of the machine is important: this allows us to de-couple a syntactic representation of a Statechart from the behavior of any particular definition of Statechart semantics. This means that we can take a Statechart created in one tool and simulate and analyze it using the execution semantics of an entirely different tool.

We now describe the translation of individual Statechart features into Java code. The entire generated code for a Statechart consists of all of these individual pieces, but we focus on one feature at a time for clarity.

States. States are the basic unit of a Statechart. Most Statechart languages allow states to contain data and perform actions at various times, including when a state entered, when it is exited, and while the system is in that state. We represent each state as an instance of a generated Java class that extends a base class, *State*, found in the

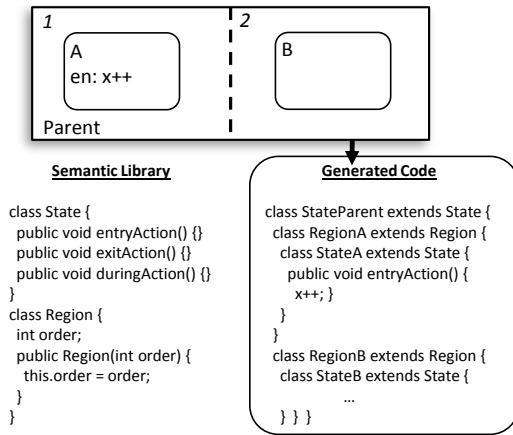


Figure 3: A parallel state and its generated code.

semantic library.

Figure 3 shows an example of how a state is translated into Java code. At the top of the Figure, we see two states, *A* and *B*. *A* contains an entry action ($x++$). On the left of the figure, we see the base class, *State*, that is found in the semantic library. The base class provides virtual methods, used by the semantic modules, that are overridden in the generated classes. The virtual methods for *State* - *entryAction*, *exitAction* and *duringAction* - allow each state to have customized behavior for these actions.

The data contained by states is translated into the corresponding data in the generated Java code. The next feature, hierarchy and parallel states, shows how generating data in this way can automatically take care of proper scoping.

Sequential and Parallel States. The top of Figure 3 shows a state *Parent* that has a parallel decomposition: its children states are *A* and *B*. To handle parallel states, we introduce the concept of a *Region*. A region is a direct concept found in the description of the UML State Machine semantics [10] that we use to represent parallel states across all variants.

Regions work as follows: for each state contained inside a given parallel state *S*, a class inheriting from the base *Region* class in the semantic library is generated inside *S*. Thus, in Figure 3, there are two classes, *RegionA* and *RegionB*, contained inside *Parent*'s generated State. The semantic modules operate such that exactly one state from each region is active at any given time.

Transitions. Transitions can have a triggering event, an optional guard which is a predicate evaluated over data values and 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.

Figure 4 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 ($x++$). The base class *Transition* in the semantic library contains four virtual methods that are overridden in the generated code to implement this functionality. Evaluation of the triggering condition is done by string comparison. There is both an *action* method and a *conditionAction* method. This is an example of a feature contained in a base class inside the semantic library that is not used by all Statechart variants.

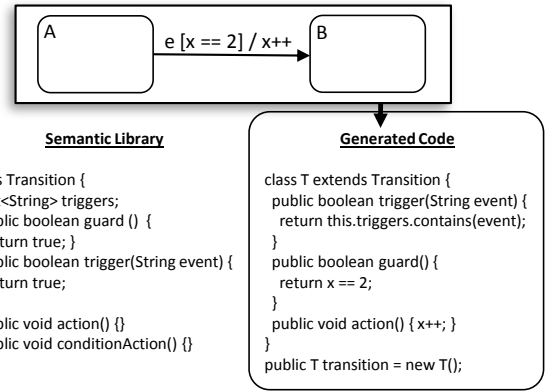


Figure 4: A transition and its generated code.

In this case, transitions using Stateflow semantics can have both types of actions while transitions using UML State Machine semantics contain only an *action* method.

Pseudostates. *Pseudostates* are additional syntax in Statecharts to reduce and simplify transitions within charts. Pseudostates represent transient locations within a state machine; a state machine cannot be resident in a pseudostate at the end of a step. Pseudostates supported in statecharts include *choice* for branching and *initial* pseudostates to describe the default value of a state upon entry (the clear and filled circles in Figure 1, respectively). Because of pseudostates, transitions may be composed of several *transition segments*, one segment for each destination (state or pseudostate) visited during evaluation of a transition. Customized 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, such as junction or choice, is given by an enumeration value. A pseudostate's kind is examined by the semantic library during chart execution.

3.2 Input-Output Interface

We now describe the input-output interface to our generated code. This feature allows the generated Java code representing a Statechart to be driven manually by the user, non-deterministically by JPF, or symbolically by Symbolic Pathfinder (SPF). For simplicity, we focus on the description of the input interface; the output interface is similar. We also focus primarily on the interface to SPF to demonstrate how our tool can be used to generate test vectors for different semantic variants of Statecharts.

Figure 5 shows the overall structure and workflow of the framework. We describe this Figure in detail to show how the individual pieces fit together. The top of Figure 5 (labeled 1) shows a Statechart with two input variables and two output variables. The inputs are an integer x and a boolean b , and the outputs are an integer y and a boolean c . There is a default transition to state *A*, along with two additional transitions. One transition goes from state *A* to state *B* and has a guard that is satisfied when the value of the input variable x is greater than 0. If this transition is taken, then it sets the value of the output variable y to the value of x . The transition from state *B* to state *A* has a guard that is satisfied when (1) the value of the output variable y is greater than 2, and (2) the value of the input

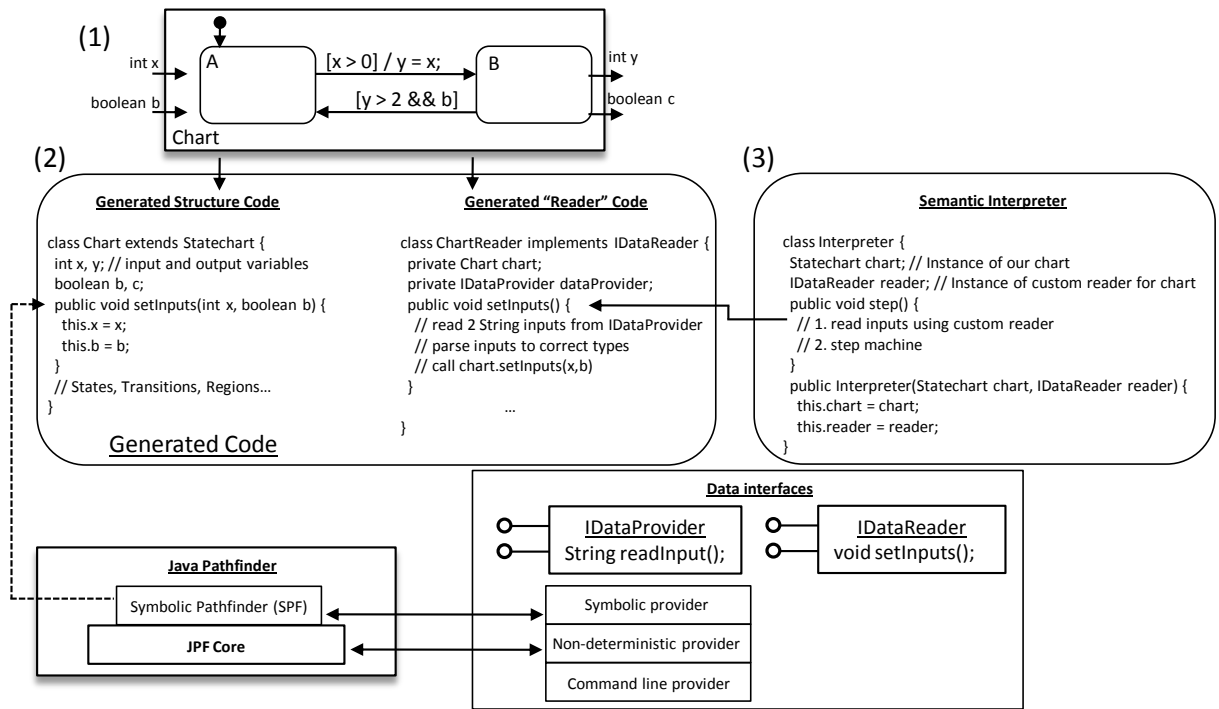


Figure 5: The interface to the Java code.

variable b is true.

The first part of our process translates the structure of this Statechart into Java code. The translation for individual elements such as states and transitions is described earlier in Section 3 and is not shown here. The second part of Figure 5 (labeled 2) shows the code that is generated for: (1) the Statechart, and (2) functionality to provide inputs to the Statechart in both a generic in specific manner. The generated structure code for the Statechart contains four variables corresponding to the inputs and outputs of the original model. Additionally, it contains a specific method for setting the values of the inputs: `setInputs(int x, boolean b)`. However, because this method is specific to a Statechart that takes exactly two inputs of a specific type, it cannot be used in a generic way. For this reason, we generate a separate module that does two things: (1) exposes a generic interface to read inputs from various sources, and (2) uses the generated type-specific method of the chart to set its inputs.

3.3 Sample Execution with SPF

We describe now how the whole execution environment can be used with SPF to generate test-vectors for the chart in Figure 5. The semantic interpreter (shown on the right in Figure 5) contains an instance of our generated `Chart` class and an instance of our generated `ChartReader`. At each logical step in the chart execution, the interpreter (1) sets the inputs of the machine using the `IDataReader`, and (2) uses the current state configuration and data valuations, along with the machine structure, to step the machine according to a specific type of semantics (Stateflow, Rhapsody or UML State Machine semantics). The semantics are described briefly here and formally in Section 4.

Suppose we have a symbolic data provider, the Stateflow

execution semantics, and that we are currently in state A . When the interpreter calls its `step()` method to advance the chart, it first calls the `setInputs()` method on its data reader object. Because the data reader is using a symbolic data provider, input does not come from the user or non-deterministically from JPF. Instead, SPF treats the inputs to method `setInputs(int x, boolean b)`, *symbolically*. When control is returned to the interpreter to step the machine, it checks if there are any enabled transitions. When the interpreter calls the `guard()` method on the transition from state A to state B (which returns `true` if x is greater than 0), SPF explores two possibilities for x : one that will cause the method to return `true`, and another that will cause the method to return `false`. For the `true` case, SPF will collect $x_1 > 0$ in the path condition PC , where x_1 is the symbolic value of input x . After the transition is taken, its action ($y=x$) is performed.

The process will then repeat with current state B : the interpreter will call the `setInputs(int x, boolean b)` method which will again be executed symbolically. When the interpreter calls the `guard()` method on the transition from B to A , condition $y > 2 \ \&\& \ b$ is evaluated. In the `true` case, because y was set to x_1 in the previous step, the PC becomes $x_1 > 0 \wedge x_1 > 2 \wedge b_2$, where b_2 is the symbolic value of b and time step 2. After determining that this constraint is satisfiable, SPF has determined that the guard on the transition from B to A is satisfiable and that there is a binding of values to the symbolic variables that allows the method to return `true`. The interpreter then takes the transition from B to A and updates the current state set.

In effect, we have used SPF to determine an input sequence of length 2 for the Statechart at the top of Figure 5 that will drive the chart from state A to state B and back to state A :

First input: $x > 2$, $b = \text{true}$ or false.
 Second input: $x = \text{anything}$, $b = \text{true}$.

SPF invokes a constraint solver to check the satisfiability of these constraints, and the solutions obtained are used as test vectors. While there exist other test-vector generation tools for Stateflow (see Section 6), the difference is that our tool can generate inputs for a given Statechart across multiple semantic variants. We believe this is an important feature needed by engineers to better understand their models.

4. PARAMETRIC FORMAL SEMANTICS

Statecharts behaviors are quite complex. For example, the user manual for Stateflow is 1400 pages and the transition semantics is described in 7 pages of pseudocode [19]. Because of this level of complexity, it is difficult to create a correct interpreter for one variant, let alone multiple variants. To ensure that our interpreters correctly describe the behavior of these variants, we have created a formal structural operational semantics (SOS) to illuminate the corner cases in each notation. The formalization has informed the interpreter implementations in Java and the selection of ‘difficult’ tests to check conformance with expected behaviors for each notation.

We believe that our work provides a more complete formalization of each variant than has been previously attempted, and have corrected errors that have been found in previous formalizations (described in more detail in Section 6). The formalization presented here is based on the formalization used for Stateflow in the Rockwell Collins tool suite described in [20]. We have identified the similarities between the semantics, and have been able to provide most of the semantics in common “core” rules. The semantic variations between the variants can be captured in a relatively small number of additional rules for each variant. Due to space limitations, it is not possible to present all the semantics of the three variants. Instead we provide a representative slice of the semantics describing transitions and refer the interested reader to an accompanying technical report [32] that contains the full rule set.

Formally, a set of SOS inference rules inductively defines the set of all computations (defined as relations) for the language. The core rules are incomplete; certain rule signatures have no definitions in the core semantics, so no complete derivations are possible. Each variant extends the core with additional inference rules that complete the semantics and define all computations in that variant.

4.1 Abstract Syntax

The abstract syntax for the parametric semantics is the union of the syntax used by each of the variants. The semantics for each variant are restricted by only defining inference rules for the syntax relevant to that variant. The abstract syntax reflects the java classes describing the structure of the chart from Section 3.1; the class definitions correspond to the datatype definitions in Figure 6.

A Statechart θ is defined as a root state named C , a set src of the different locations within the chart, and input, output, and local declarations (I , O , and L) of variables and events. Charts contain destinations v which can be states, pseudostates, and, in the case of Stateflow, graphical functions. Destinations are referenced by paths (identifier lists) p to create locations src .

```

Chart  $\theta$  ::= ( $C, [src_0, \dots, src_n], I, O, L$ )
Location  $src$  ::=  $p : v$ 
Destination  $v$  ::=  $State(sd) \mid Pseudo(psd) \mid$ 
   $Function(gfd)$ 
StateDef  $sd$  ::=  $((a_e, a_d, a_x), L, T, [r_0, \dots, r_n])$ 
Region  $r$  ::=  $(T, [s_0, \dots, s_n])$ 
PseudoDef  $psd$  ::=  $(pty, T)$ 
FunctionDef  $gfd$  ::=  $((I, O, L), T)$ 
PseudoType  $pty$  ::=  $JUNC \mid$ 
   $CHOICE \mid$ 
   $DYNAMIC_CHOICE \mid$ 
   $SHALLOW_HISTORY \mid$ 
   $DEEP_HISTORY$ 
Trans  $t$  ::=  $(e, c, (a_c, a_t), d)$ 
Dest  $d$  ::=  $p \mid LOOP$ 

```

Figure 6: Statecharts abstract syntax.

States ($StateDefs$) consist of actions associated with entering, staying within (during), and exiting the state (a_e, a_d, a_x); local variables and constants L ; a transition list T of outgoing transitions from the state; and a (possibly empty) list of regions (parallel states) R . *Regions* contain a sequential composition of states and a list of initial transitions T (e.g., the transitions starting with a dot in Figure 1).

Pseudostates ($PseudoDefs$) such as history states, static and dynamic choice points, have an associated type and a list of outgoing transitions. We do not include a complete set of UML/Rhapsody pseudostate types in our definition. The types that are missing are: *initial*, *join*, *fork*, *entryPoint*, *exitPoint*, and *terminate*. We handle *join* and *fork* via a source-to-source translation for Rhapsody and UML Statecharts using a variation of the encoding used by Börger [4]. *Initial* pseudostates are encoded directly via the set of initial transitions for compositions. *EntryPoint* and *exitPoint* pseudostates are currently unsupported in our semantics, as they are syntactic sugar (see [4]); *terminate* pseudostates are currently unsupported, as they require knowledge about the object containing the statemachine that is not currently defined in the semantics. *FunctionDefs*, defined in Stateflow, allow graphical functions to be created from transitions.

A transition $Trans$ contains a triggering event e (which is the reserved symbol \perp in the case of an eventless transition in Stateflow or a completion transition in UML Statecharts or Rhapsody), guarding condition c , condition and transition actions (a_c, a_t) and destination d . A destination is a path to a state, junction or the special destination *Loop* that is used for external loop transitions. Three list types $TransLst$, $path$, and $ActionLst$ describe lists of transition segments, identifiers, and actions, respectively. ϕ is the parametric symbol for the empty list (*nil*).

4.2 Overview of Semantics

The semantics of each Statecharts variant focuses around evaluation of an event. Given an event (which may be from the external environment or internally generated) the system finds the set of enabled transitions. A transition is enabled if its triggering event matches the current event and its guarding condition evaluates to true. A subset of these transitions then fire, causing the system to change state and potentially generate new events that further evolve the state machine.

In our view, the most complex part of the Statecharts semantics is the evaluation of transitions containing multiple segments, specifically: (1) how to exit and enter compound states (that is, states with child states) when transitions

‘fire’, (2) how to evaluate inter-level transitions, that is, transitions that cross state boundaries, and (3) how to evaluate compound transitions, that is, transitions containing multiple segments that are joined by pseudostates. Fortunately, this portion of the semantics is largely common between the different semantics, and the core semantics is primarily concerned with this aspect.

The differences between the variants involve (1) how and when internal events generated by the evaluation of the state machine are consumed, (2) how and when transitions without an explicit triggering event (called completion transitions in UML Statecharts and Rhapsody) are consumed, (3) given conflicting transitions, which subset are chosen to fire, and (4) whether the evaluation of the firing transitions is performed atomically or incrementally as a sequence of sub-steps. These aspects are dealt with in the parametric portion of the rule sets, and are instantiated separately for each variant. These aspects are straightforward to formalize, leading to relatively few variant rules between the different variants.

We create evaluation rules for the different pieces of syntax within the abstract syntax tree. We use turnstiles (\vdash) annotated with the kind of syntax being evaluated to structure the semantics. For example, the semantic rules for transitions and a subset of the pseudostates are shown in Table 1. In this figure, we evaluate three different classes of syntax: transition segments \vdash_τ , transition segment lists \vdash_T , and destinations \vdash_D . In the full semantics in [32], additional rules for states \vdash_S and regions \vdash_R are added. We augment the syntactic rules with helper rules and functions. Helper rules are indicated by turnstiles followed by rule names. In Table 1, the helper rules used are ‘trigger’, ‘choose’, ‘move’, ‘exit’, ‘lcp’, and ‘enter’.

A handful of conventions are assumed: first, we assume the following operations on lists: `cons` is defined infix using `::`, appending lists is defined infix as `^`, and adding an element to the end of the list is defined using a period `.'`. Second, we write rules that are specialized for each variant in **boldface**. In Table 1, the two specialized rules are the list choice rule (**choose**) and the trigger rule (**trigger**).

4.3 Actions and Conditions

Condition rules \vdash_B describe evaluation of Boolean expressions that are used for transition guards. Action rules \vdash_A describe updates to the environment that occur when a transition fires or a state is entered / active / exited. The signatures for the rules are as follows:

$$\begin{aligned}\vdash_B &\subseteq Env \times Condition \times Bool \\ \vdash_A &\subseteq Chart \times Env \times ActionList \times Env\end{aligned}$$

In this paper, these relations are left abstract; they are not difficult to express but are verbose.

4.4 Transitions

Describing the behavior of transitions in full generality is the most complex part of the semantics of Statecharts. The rules associated with transitions are shown in Table 1. Recall that transitions in Statecharts are composed of a list of transition segments that connect a source and destination state, possibly through one or more pseudostates. We will use the word *segment* to describe a single arc within a chart and *transition* to mean a complete path through a sequence of segments. The \vdash_τ and \vdash_T rules describes the behavior of a segment and list of segments, respectively. Elements

of T represent the list of outgoing segments from a state or pseudostate. The \vdash_D rules describe the behavior of the chart upon reaching the *destination* of the segment, which can be either a state or a pseudostate.

The signatures of the different rule types associated with transitions is shown below:

$$\begin{aligned}\vdash_\tau &\subseteq Chart \times Env \times (Path\ list) \times \\ &\quad (Action\ list) \times Trans \times (Env, Status) \\ \vdash_T &\subseteq Chart \times Env \times (Path\ list) \times \\ &\quad (Action\ list) \times (Trans\ list) \times (Env, Status) \\ \vdash_D &\subseteq Chart \times Env \times (Path\ list) \times \\ &\quad (Action\ list) \times Dest \times (Env, Status)\end{aligned}$$

The rules are evaluated in a context containing the chart function ($\theta : Chart$) the current environment ($\sigma : Env$), the list of destinations ($P : Path\ list$) and transition actions ($A_t : Action\ list$) encountered along the current transition path. The chart θ is used to look up destinations within the chart (states and pseudostates), and the environment σ contains the current values of all elements in the chart (the occupied/unoccupied status of states and values of chart variables). The rules generate a new environment and a status which can be either **Fail** if the current path does not reach a destination state, **Succeed(State)** if the path completes a transition to a destination state.¹

For example, the rule τ_1 describes the case in which a segment ‘fires’. The rule premises state that the trigger is present, the condition evaluates to true, the *condition actions* associated with the transition modify the environment to σ' , and the evaluation of the destination of the segment d given new environment σ' and the updated set of *transition actions* ($A_t \frown a_t$) returns the result *res*. If these premises are satisfied, then the evaluation of the segment $(e_t, c, (a_c, a_t), d)$ will return *res*. Rules τ_2 and τ_3 describe the situation when the trigger and the guarding condition are false, respectively.

4.5 Transition Lists

There are three rules to evaluate transition lists (T_1, \dots, T_3). Rule T_1 describes the case when the list of transitions is empty, so we return that this path failed to yield a complete transition (**Fail**). In rule T_2 , we choose an element from the transition list, and it completes a transition, so we return its result. In rule T_3 , we choose an element from the list but it does not lead to a complete transition, so we evaluate the rest of the list recursively. Note that a segment that does not succeed may still modify the environment σ .

The **choose** rule is an example of a variation point between the semantics: in Stateflow, the order of evaluation of segments from a particular source is fixed, so the first element of this list is always the first element chosen (rule cs_1). For Rhapsody and UML Statecharts, **choose** can choose an arbitrary element from the list (rules cr_{u1} and cr_{u2}).

4.6 Destinations

The destination rules D define the behavior of the chart when a ‘fired’ transition reaches a destination. As we evaluate segments along a path, we build up a list of transition actions A_t and visited destinations P in the transition. The

¹Stateflow has an additional completion: **Succeed(Junction)** if the path reaches a junction with no outgoing transitions

$\tau_1 \frac{\theta, \sigma \vdash \text{trigger } e_t \rightarrow \text{true} \quad \theta \vdash_B c \rightarrow \text{true} \quad \theta, \sigma \vdash_A a_c \rightarrow \sigma' \quad \theta, \sigma', P, (A_t \frown a_t) \vdash_D d \rightarrow \text{res}}{\theta, \sigma, P, A_t \vdash_T (e_t, c, (a_c, a_t), d) \rightarrow \text{res}}$	
$\tau_2 \frac{\theta, \sigma \vdash \text{trigger } e_t \rightarrow \text{false}}{\theta, \sigma, P, A_t \vdash_T (e_t, c, (a_c, a_t), d) \rightarrow (\sigma, \text{Fail})}$	$\tau_3 \frac{\theta \vdash_B c \rightarrow \text{false}}{\theta, \sigma, P, A_t \vdash_T (e_t, c, (a_c, a_t), d) \rightarrow (\sigma, \text{Fail})}$
$T_1 \frac{}{\theta, \sigma, P, A_t \vdash_T \phi \rightarrow (\sigma, \text{Fail})}$	$T_2 \frac{\text{choose}(t :: L) \rightarrow (\tau, \text{rest}) \quad \theta, \sigma, P, A_t \vdash_T \tau \rightarrow (\sigma', \text{Succeed}(dt))}{\theta, \sigma, P, A_t \vdash_T (t :: L) \rightarrow (\sigma', \text{Succeed}(dt))}$
$T_3 \frac{\text{choose}(t :: L) \rightarrow (\tau, \text{rest}) \quad \theta, \sigma, P, A_t \vdash_T \tau \rightarrow (\sigma', \text{Fail}) \quad \theta, \sigma', P, A_t \vdash_T \text{rest} \rightarrow \text{res}}{\theta, \sigma, P, A_t \vdash_T \phi \rightarrow \text{res}}$	
$D_{s1} \frac{\theta \text{ path} = \text{State}(S) \quad \theta, \sigma, A_t \vdash \text{move}(P.\text{path}) \rightarrow \sigma'}{\theta, \sigma, P, A_t \vdash_D \text{path} \rightarrow (\sigma'', \text{Succeed}(\text{State}))}$	$D_{s2} \frac{\theta, \sigma, A_t \vdash \text{move}[p.s, p, p.s] \rightarrow \sigma'}{\theta, \sigma, [p.s], A_t \vdash_D \text{LOOP} \rightarrow (\sigma'', \text{Succeed}(\text{State}))}$
$D_{p1} \frac{\theta \text{ path} = \text{Pseudo}(\text{JUNC}, \phi)}{\theta, \sigma, P, A_t \vdash_D \text{path} \rightarrow (\sigma'', \text{Succeed}(\text{Junction}))}$	$D_{p2} \frac{\theta \text{ path} = \text{Pseudo}(\text{JUNC}, h :: t) \quad \theta, \sigma, P.\text{path}, A_t \vdash_T (h :: t) \rightarrow \text{res}}{\theta, \sigma, P, A_t \vdash_D \text{path} \rightarrow \text{res}}$
$D_{p3} \frac{\theta \text{ path} = (\text{Pseudo}(\text{CHOICE}, h :: t) \quad \theta, \sigma, P, A_t \vdash_T (h :: t) \rightarrow \text{res}}{\theta, \sigma, P, A_t \vdash_D \text{path} \rightarrow \text{res}}$	
$\text{move} \frac{\text{lcp}(src :: (P.dst)) = cp \quad \theta, \sigma \vdash \text{exit}(src, cp) \rightarrow \sigma' \quad \theta, \sigma' \vdash_A A_t \rightarrow \sigma'' \quad \theta, \sigma'' \vdash \text{enter}(dst, cp) \rightarrow \sigma'''}{\theta, \sigma \vdash \text{move}(src :: (P.dst)) \rightarrow \sigma'''}$	
Stateflow: $cs_1 \frac{}{\text{choose}(h :: t) \rightarrow (h, t)}$	UML/Rhapsody: $cru_1 \frac{}{\text{choose}(h :: t) \rightarrow (h, t)} \quad cru_2 \frac{\text{choose } t \rightarrow (e, r)}{\text{choose}(h :: t) \rightarrow (e, h :: r)}$

Table 1: Statecharts transition rules.

first rule D_{s1} describes the behavior when the destination is a state. In this case, we call the ‘move’ helper rule (near the bottom of the figure) to change from the source to the destination state, and the rule returns $\text{Succeed}(\text{State})$ to denote that the transition has reached a destination.

Rule D_{s2} is used for external loop transitions. There are two kinds of looping transitions allowed in Statecharts: internal loop and external loop transitions. Because they share the same source and destination, but behave differently (internal loops do not cause the exit of the state, external loops do), we must treat one of them specially. Our ‘move’ rule extracts the common parent of the elements along the transition path to determine which states to be exited/entered. Because a transition from a state to itself is entirely common, it causes exit / reentry of only child states (i.e., internal loop). To describe external loops, we use a special ‘LOOP’ destination, and construct a path that goes from the state containing the loop to its parent and back. This will cause the correct entry/exit behavior.

Rules D_{p1} through D_{p3} describe the behavior of the rules for *JUNC* and *CHOICE* pseudostates. In [32], we define the behavior of the remaining pseudostate types in the abstract syntax.

The ‘move’ rule describes the updates to the state performed during a move from a source to a destination state. The list of locations involved is $(src :: (P.dst))$, that is, the source state, followed by a (possibly empty) list of pseudostates P followed by the destination state dst . We find the least common parent (cp) of the list to determine the scope of the transition (the set of states to be entered and exited). We then (1) exit the source component, (2) perform the transition actions A_t from the path, and (3) enter the destination state.

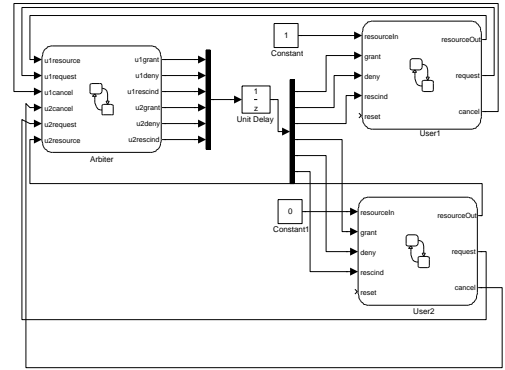


Figure 7: Model of MER arbiter with two users

4.7 State and Composition Rules

Given the definition of transition behavior, we can define the rules for states and regions. Because of space considerations, we do not present the rules here, but refer the interested reader to the tech report [32]. The rules for states and compositions are simpler than those for transitions. The main point of interest is the variation between the semantics in transition prioritization: UML Statecharts and Rhapsody prioritize transitions ‘inside out’, giving priority to transitions from nested states, whereas Stateflow prioritizes transitions in the opposite ‘outside in’ order.

5. EXPERIENCE

The framework and tools described in this paper have been applied to an example modeling a component of the

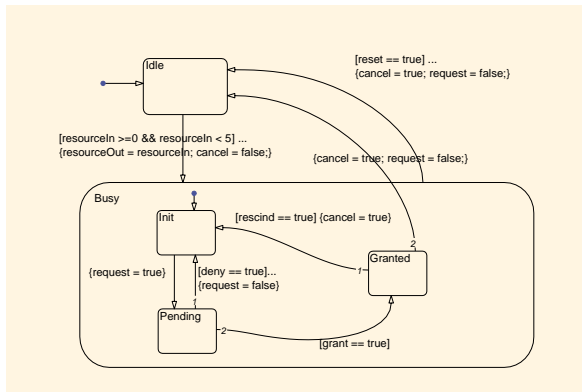


Figure 8: State machine for user 1

flight software for NASA/JPL’s Mars Exploration Rovers (MER). The MER software consists of a Resource Arbiter and several user threads. Each user serves one specific application, such as imaging, controlling the robot arm, communicating with earth, and driving. The arbiter module moderates access to several shared resources. It prevents potential conflicts between resource requests coming from different users and it enforces priorities. For example, it does not make sense to start a communication session with Earth while the rover is driving.

We present here the results of modeling and analyzing the system using multiple Statechart formalisms; the system has been analyzed before, but using only one formalism (namely Promela) [23]. The configuration for our analysis involved two users and five resources. The high-level architecture of the analyzed system is given in Figure 7. The system was first modeled in Simulink, with the two users and the arbiter being modeled as Stateflow machines. The inputs to the system are the two *resources* and the two *resets* that go to the two users. The communication between the Arbiter and the users is modeled using Simulink signals. A user may *request* or *cancel* a resource; the arbiter may *grant* or *deny* the resource, and it can also *rescind* the resource after it has been granted. Figure 8 shows the State machine for one user.

This Simulink/Stateflow model was translated automatically into Java and analyzed with SPF. The individual Stateflow models were translated as described in Section 3. We used the tool described in [24, 3] to automatically translate the Simulink part that connects the individual Stateflow components. For now, this is the mechanism that we use to assemble components. NASA Exploration engineers are still defining how components should be assembled. We are working on a library of “connectors” that provides a Java representation of various types of communications between components (see also discussion in Section 7).

We have experimented with checking safety properties and generating test cases for this model, where we changed the semantics of User 1 from Stateflow into UML and Rhapsody (while User 2 and the Arbiter were kept with the Stateflow semantics). As mentioned, changing the semantics can be easily achieved in our framework (all one needs to do is to change the `Interpreter` for that machine). We inserted `assert(r1.User1.reset==false)` on the transition from state `Pending` to `Granted` in User 1 and we checked for assertion violations in the over-all system. This encodes the expecta-

tion that once a reset is received, a component should not be able to start using the resource. To study the behavior of the models with increased number of time steps, we have analyzed three versions of the models, corresponding to sequences of sizes 4, 5 and 6.

The results of our analysis are summarized in Table 2.

We observe that the property holds for the Stateflow models, but it fails when we change the semantics of one user to UML or Rhapsody. In this latter case, SPF generates the following 2-step test case that exposes the error.

```
setUser1Input(1,0),
setUser2Input(2,1),
```

```
setUser1Input(3,1),
setUser2Input(2,0),
##EXCEPTION## "java.lang.AssertionError..."]
```

The reason why this property fails in the UML and Rhapsody cases while it holds in the Stateflow case is that outer transitions (e.g. see the transition enabled on `reset==true` from `Busy` back to `Idle`) have higher priority over inner transitions in Stateflow, but have lower priority in UML Rhapsody. The semantic difference between Stateflow on one side and UML and Rhapsody on the other side is also reflected in the different number of test cases generated for one semantic variant over the other. We also note that the results for UML and Rhapsody are practically identical. This is expected, since the main difference between Rhapsody and UML is that the result of an action is immediately visible in Rhapsody, but that doesn’t affect our particular example.

This example demonstrates how we can use Polyglot to model components using multiple Statechart formalisms and to analyze their inter-operation. It also shows how properties can be checked with our techniques and how we can generate test cases that expose semantic difference across models. We note that the generated test cases can be used for testing the code that is generated (automatically or manually) from the models.

6. RELATED WORK

There are at least dozens and perhaps hundreds of different Statecharts variants [30]. A handful of these variants were created with a formal semantics, including Esterel SyncCharts [2], RSML [16], and RSML-e [31], but for the most part they are defined informally; semantics have been ascribed to these notations after the fact. The three variants considered here, Stateflow, UML State Machines and Rhapsody, all fit in the category of informally defined semantics but are amongst the most widely used in practice.

There are several formalizations of UML Statecharts using Abstract State Machines (ASM). The most complete formalization for the single chart case is [4]. Börger provides a nice, modular description of the UML semantics. The treatment in Börger does not match the stated semantics of [10] in an important way, however, involving the interleaving of actions and conditions. In [10] the set of all transitions to be executed is considered prior to any actions on any of those transitions. However, in Börger, the sequence of evaluation is interleaved. For example, when evaluating the model in Figure 1 using the semantics in [4] the interpretation will match the Rhapsody semantics rather than the UML semantics. This interleaving is also present in Börger’s proposal for boundary crossing transitions into AND-states.

Table 2: Experimental results

Semantics, Seq. size	Total # Test Cases	Property	Memory, Time
U1 Stateflow, 4	125	true	20 M, 43 s
U1 Stateflow, 5	412	true	22 M, 2 m 04 s
U1 Stateflow, 6	1343	true	24 M, 6 m 46 s
U1 UML, 4	57	false	21 MB, 21 s
U1 UML, 5	155	false	21 MB, 53 s
U1 UML, 6	579	false	23 MB, 2 m 50 s
U1 Rhapsody, 4	57	false	21 MB, 21 s
U1 Rhapsody, 5	155	false	21 MB, 55 s
U1 Rhapsody, 6	579	false	23 MB, 2 m 45 s

Semantics for smaller subsets of UML Statecharts are provided using several different formal frameworks by several groups. Compton et al [6] uses ASMs, Von der Beek [29] uses SOS rules somewhat similar to ours, Gogolla et al. [9] uses translation to an abstract machine using graph rewriting and Reggio et al. [27] uses translation to the algebraic specification language CASL. There is no work that we are aware of that specifically formalizes Rhapsody semantics.

Stateflow semantics have been formalized by Hamon twice: once as an operational semantics [13] and later as a denotational semantics [12]. As mentioned earlier, we base our operational semantics on the denotational rather than the operational semantics. The reason for this choice is both that the denotational semantics is more complete and more modular, and there was little difficulty in moving from Hamon’s continuations to an alternate backtracking form that explicitly returned success or failure. Moreover, we have fixed several small errors in Hamon’s semantics regarding (1) Transition action sequencing, (2) correct state entry on AND-state boundary-crossing transitions, (3) scoping on multi-segment boundary crossing transitions, (4) flowcharts in states with no substates, (5) border-crossing initial transitions, (6) ‘during’ actions with internal transitions, and (7) entry/exit actions associated with external loop transitions.

Additionally there is a body of work that ascribes semantics to multiple Statecharts variants. Perhaps the most ambitious is the template-based semantics of Jianwei Niu et. al [21] that provides a generic template semantics for many Statecharts variants as well as process algebras. Although the template mechanism in [21] is quite flexible, it is not sufficient to describe the behavior of any of the three notations considered here. In particular, there is no distinction between states and pseudostates, so it is not possible to describe the different behaviors (such as backtracking and looping in Stateflow junctions and dynamic vs static choices in UML) that distinguish states from pseudostates. Also, there is no mechanism to distinguish condition actions from transition actions (as in Stateflow), so it is not possible to correctly model (one of) condition or transition actions.

There are many tools for analyzing Stateflow models due to the widespread use of Simulink/Stateflow, including commercial tools such as Mathworks’ *Design Verifier* which can be used for model checking and test case generation, and Reactive System’s *Reactis* and T-VEC’s *tester* that perform test generation and measurement. Similarly, for UML Statecharts, there are a wide variety of research tools. However, we believe that the ability to analyze multiple semantics in one environment is a major benefit to our approach.

One idea similar to our unified environment is found in [25], which describes an approach to heterogeneous model

analysis that is based on a common “inframodel” that captures only the aspects of a notation essential to state space exploration and it provides a set of rules that capture the semantics and interactions between multiple formalisms. The work is concerned with high-level descriptions of the models and it would take considerable effort to make that environment capture the semantic details for the Statechart notations that are the focus of our work. Therefore, using that work one can not generate detailed test cases, that are useful for testing the low-level code that was generated from the models. Furthermore that work does not address checking that a model preserves the same properties when run with different semantics.

The Ptolemy environment [8] is a laboratory for experimenting with different models of computation for component based systems, focusing on the concurrency and temporal issues. The tool implements the concept of polymorphic components whose behavioral semantics depends on what ‘execution engine’ (‘director’ in Ptolemy) it is executed by. This is similar to our concept of ‘pluggable semantics’. Note that our work addresses different Statechart variants and formal semantics with particular focus on model checking and systematic test case generation, while Ptolemy addresses a broader set of models of computation but with the goal of simulation.

In previous work [24] we have developed a model based analysis framework that translates Simulink/Stateflow models into Java and analyzes them using Java PathFinder; that work was recently extended with Specification Patterns for writing and checking temporal logic formulas [3]. In that work we use model transformation techniques to translate Simulink/Stateflow into Java; the translation follows very much the structure of the Mathworks’ Real-Time Workshop code generation. Here we have developed a completely new translator for the Stateflow parts together with new translators for UML and Rhapsody. The obtained translated Java code is very different, to allow for pluggable-semantics described in this paper.

7. CONCLUSION

We have presented Polyglot, a framework for the analysis and test case generation of model-based flight software described using multiple Statechart formalisms. In Polyglot, multiple variants of Statecharts are translated into a common Java representation with “pluggable” semantics for different Statechart variants.

To provide confidence in our framework, we developed a parameterized description of the semantics of Statecharts across several of the most popular variants in structural operational semantics. We have indicated improvements over

previous formalizations. We have also discussed the application of Polyglot to the analysis of the MER Arbiter system.

In the future, we plan to extend the communication infrastructure of our component-based framework to facilitate component interactions. We plan to provide an extensible library of connectors that capture different types of communication policies. For example, connectors may model procedure calls with call-backs, event-based synchronization, as well as domain-specific protocols and standards such as the ARINC 653, an RTOS API specification with support for space and time partitioning in an integrated modular avionics architecture.

We are also working on modeling the semantics of the Plexil robot execution system [28] using a variant of Statecharts and on integrating it in the Polyglot framework. The goal is to expand Polyglot into a *heterogeneous* modeling and analysis environment useful for NASA engineers.

8. REFERENCES

- [1] Java pathfinder tool-set. <http://babelfish.arc.nasa.gov/trac/jpf>, 2011.
- [2] C. André. Computing synccharts reactions. *Electron. Notes Theoretical Computer Science*, 88:3–19, 2004.
- [3] D. Balasubramanian, G. Pap, H. Nine, G. Karsai, M. Lowry, C. Pasareanu, and T. Pressburger. Rapid property specification and checking for model-based formalisms. In *submitted for publication*, 2011.
- [4] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of uml state machines. In *ASM '00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, pages 223–241, London, UK, 2000. Springer-Verlag.
- [5] A. Childs, J. Greenwald, V. P. Ranganath, X. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, P. Shanti, and G. Singh. Cadena: An integrated development environment for analysis, synthesis, and verification of component-based systems. In *FASE*, pages 160–164, 2004.
- [6] K. Compton, J. Huggins, and W. Shen. A semantic model for the state machine in the unified modeling language. In *In Proceeding of Dynamic Behavior in UML Models: Semantic Questions, UML 2000 workshop*, pages 25–31. Springer Verlag, 2000.
- [7] M. L. Crane and J. Dingel. Uml vs. classical vs. rhapsody statecharts: Not all models are created equal. 2005.
- [8] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.
- [9] M. Gogolla and F. Parisi-Presicce. State diagrams in uml: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, 1998.
- [10] O. M. Group. *Unified Modeling Language™ (OMG UML), Superstructure Version 2.2*. Object Management Group, February 2009.
- [11] M. Hamilton. The heart and soul of apollo: Doing it right the first time. In *Proc. 7th International Military and Aerospace Programmable Logic Devices (MAPLD) Conference*, 2004.
- [12] G. Hamon. A denotational semantics for stateflow. In *Proceedings of the Embedded Systems Software Conference*, Jersey City, New Jersey, September 2005. ACM.
- [13] G. Hamon and J. Rushby. An operational semantics for stateflow. In *Proceedings of the Fundamental Approaches to Software Engineering (FASE) Conference*, pages 229–243, Barcelona, Spain, March 2004. Springer Verlag (LNCS 2984).
- [14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [15] D. Harel and H. Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In *In Integration of Software Specification Techniques for Application in Engineering, number 3147 in Lecture Notes in Computer Science*, pages 325–354. Springer, 2001.
- [16] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
- [17] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
- [18] Á. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.
- [19] Mathworks Inc. Stateflow product web site. <http://www.mathworks.com>.
- [20] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.
- [21] J. Niu, J. Atlee, and N. Day. Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, 29(10):866 – 882, October 2003.
- [22] C. Pasareanu and N. Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of ASE*, pages 179–180, 2010.
- [23] C. S. Pasareanu and D. Giannakopoulou. Towards a compositional spin. In *Proceedings of SPIN Workshop*, 2006.
- [24] C. S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema. Model based analysis and test generation for flight software. In *SMC-IT*, 2009.
- [25] M. Pezzè and M. Young. Constructing multi-formalism state-space analysis tools: Using rules to specify dynamic semantics of models. In *ICSE*, pages 239–249, 1997.
- [26] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.
- [27] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing uml active classes and associated state machines - a lightweight formal approach. In *Fundamental Approaches to Software Engineering, LNCS 1783*, pages 127–146. Springer, 2000.
- [28] V. Verma, T. Estlin, A. Jonsson, C. S. Pasareanu, and R. Simmons. Plan execution interchange language (plexil) for command execution. In *Proceedings of iSAIRAS*, 2005.
- [29] M. von der Beeck. A structured operational semantics for uml-statecharts. *Software and Systems Modeling*, 1:130–141, 2002. 10.1007/s10270-002-0012-8.
- [30] M. von der Beeck. A comparison of statecharts variants. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 526, 1994.
- [31] M. W. Whalen. A formal semantics for $RSML^{-e}$. Master's thesis, University of Minnesota, May 2000.
- [32] M. W. Whalen. A parametric structural operational semantics for stateflow, uml statecharts, and rhapsody. Technical Report 2010-1: <http://www.umsec.umn.edu/publications>, University of Minnesota Software Engineering Center, 200 Union St., Minneapolis, MN 55455, August 2010.