Improving Symbolic Execution for Statechart Formalisms

Daniel Balasubramanian  
Vanderbilt University  
Nashville, TN  
daniel@isis.vanderbilt.edu

Corina Păsăreanu  
NASA Ames Research Center  
Mountain View, CA  
corina.s.pasareanu@nasa.gov

Michael W. Whalen  
University of Minnesota  
Minneapolis, MN  
whalen@cs.umn.edu

Gábor Karasi  
Vanderbilt University  
Nashville, TN  
gabor@isis.vanderbilt.edu

Michael Lowry  
NASA Ames Research Center  
Mountain View, CA  
michael.r.lowry@nasa.gov

ABSTRACT
Symbolic execution is a program analysis technique that attempts to explore all possible paths through a program by using symbolic values rather than actual data values as inputs. When applied to Statecharts, a model-based formalism for reactive systems, symbolic execution can determine all feasible paths through a model up to a specified bound and generate input sequences exercising these paths. The main drawback of this method is its computational expense. This paper describes two efforts to improve the performance of symbolic execution within our previously developed framework for Statechart analysis. One method is a multithreaded symbolic execution engine targeted directly at our framework. A second, orthogonal, method is program specialization with respect to a particular model and Statechart semantics, which uses symbolic execution to rewrite the original code into an equivalent form that has fewer instructions and is easier to analyze.

Categories and Subject Descriptors
D.2.5 [Testing and Debugging]: Symbolic execution

General Terms
Design, languages, verification

Keywords
Symbolic execution, analysis, Statecharts, test case generation.

1. INTRODUCTION
Since their introduction, Statecharts [5] have become one of the most widely used model-based notations for describing complex, reactive systems. There are literally dozens of variants of Statecharts, each having a similar but slightly different execution semantics [3]. One example where multiple semantic variants of Statecharts have been used together to model different, interacting components of a large system is the Constellation program at NASA. While the use of multiple tools is convenient for developers, component integration and analysis can be difficult due to different semantics between the tools.

One form of analysis that is well-suited to Statecharts is symbolic execution [6]. Symbolic execution is a program analysis technique that systematically explores different program paths by using symbolic values rather than actual data values as inputs and computing the values of program variables as symbolic expressions in terms of the symbolic inputs. When applied to Statechart formalisms, symbolic execution can determine all feasible paths through a model up to a user-specified bound and perform test-case generation by generating sequences of inputs that drive a model through those paths.

However, there are two major hurdles to using symbolic execution to analyze Statechart models. First, a suitable symbolic execution engine, which can execute a model symbolically, must be integrated with the modeling framework. Second, symbolic execution can be computationally expensive when analyzing large, complex pieces of code.

In our previous work, we addressed the first problem by developing Polyglot [2], a unified framework in which interacting components modeled with different Statechart variants can be simulated and analyzed. Polyglot works by translating the structure of a model into Java code, and then combining this structure code with pluggable modules that implement the semantics of different variants of Statecharts. To perform analysis, Polyglot uses Symbolic PathFinder (SPF) [7], a symbolic execution engine built on top of Java PathFinder (JPF)'s custom virtual machine [1]. SPF generates path coverage tests up to some depth; these could be used to test a Statecharts compiler or a model-based implementation. While this approach scales for small to medium sized examples, the performance of SPF on large models can be slow, due to the interpretation overhead of the custom virtual machine.

This paper describes two contributions that improve the performance of symbolic execution for Statechart formalisms. The first is a multithreaded symbolic execution engine for Polyglot. Even though the symbolic execution of Statecharts and multithreaded symbolic execution have been described
elsewhere independently ([9] and [8], resp), our new framework provides some advantages, such as the ability to handle both hierarchy and parallel states. Additionally, the design of Polyglot, which decouples the semantic modules from the "structure" of a Statechart model, readily lends itself to a multithreaded implementation.

The second complementary contribution is the application of program specialization to the generated Polyglot code for particular models and particular semantics. The approach is akin to partial evaluation [4] and it uses SPF to first examine the model in Polyglot and then to generate a specialized version whose semantics are equivalent to the original. The specialized code is simplified with respect to a Statechart model, resulting in fewer instructions to be executed and analyzed by SPF when further analyzing or simulating the model.

2. BACKGROUND

2.1 Polyglot

Figure 1 gives an overview of the Polyglot framework[2]. Components can be defined inside the Simulink/Stateflow or Rational Rhapsody environments (top-left corner of Figure 1) and automatically imported into an intermediate representation. From the intermediate representation, Java code representing only the structure of the model, not the semantics, is generated (top-right corner of Figure 1). In this way, the semantics of a particular Statechart variant are decoupled from the structure of the model, allowing, for instance, different models to be simulated and executed with multiple semantics. To execute a model, the generated structure code is combined with a semantic module, currently for either Stateflow, UML or Rhapsody semantics. To perform analysis, the semantic modules expose a generic data interface that SPF uses to interact with and control the symbolic execution of a model.

One important feature of the Polyglot design is the decoupling of the structure of a Stateflow model from its semantics. Figure 1 shows that the structure of a model is turned into Java code and combined with the semantic modules to provide an execution. With respect to the "state" of a model, the semantic modules hold only a list of currently active states. This design allows the semantic modules to act as interpreters of the generated structure code: a module can inspect and set the variable values for a model while storing the list of active states internally. Section 3.3 describes how this case our multithreaded implementation.

2.2 Symbolic execution

Symbolic execution [6] is program analysis technique that uses symbolic values instead of actual data as inputs and uses symbolic expressions to represent the values of program variables. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. 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, collecting the conditions on the inputs along the explored paths. The paths followed during the symbolic execution of a program are typically characterized by a symbolic execution tree.

3. CUSTOM SYMBOLIC EXECUTION FOR POLYGLOT

This section describes the design and implementation of a multithreaded symbolic execution engine for Polyglot. As mentioned previously, using Symbolic PathFinder (SPF) to analyze Statechart models in the Polyglot framework can be computationally expensive. In general, symbolic execution may be expensive due to the large number of paths to explore. Furthermore SPF performs symbolic execution by implementing a non-standard interpreter of Java bytecodes inside JPF’s custom virtual machine. The use of the custom virtual machine alone incurs significant interpretation overhead. In addition, the execution of each transition step in the model involves invoking and executing several Java data-structures and methods that provide the implementation in Polyglot; in other words, SPF is symbolically executing the execution engine that executes the model, and this has a much larger state space than the model itself.

In order to remove the overhead of SPF, we designed and implemented a custom symbolic execution engine targeted directly at models in the Polyglot framework. For our initial effort, we refactored the Stateflow semantic module from a concrete execution engine into a symbolic execution engine. Additionally, we designed this engine with multithreading in mind, so that analysis can be distributed across multiple cores. The rest of this section describes the design and implementation, using a small example to explain the involved concepts and algorithms. Even though this presentation is done in the context of Stateflow semantics, we believe this approach generalizes, and that our methods can be used as a "recipe" for constructing symbolic execution engines for other Statechart like modeling languages as well.

3.1 Running example

The example Statechart model that will be used to illustrate the concepts throughout this section is shown in the top left-hand corner of Figure 2. This model has four states (A, B, C and D), four transitions (t0, t1, t2 and t3) and two integer data variables: an input variable x and an internal variable a. Transition t1 is labeled with a condition action, meaning that when transition t1 is taken, a is assigned the value x + 1. Transition t2 and t3 each have a guard, which is a condition that must be true for the transition to be valid. The guard on t2 states that the value of the variable a must be greater than zero, while the guard on t3 states that the value of a must be less than zero.
Execution of this model with Stateflow semantics works in the following way. Initially, assume the model is in state A and the value of a is zero. At the beginning of every step of execution, the value of the input variable x is read from the environment. Next, transition t1 is tested, and because it has no guard, t1 is taken, its condition action performed, and the current state becomes state B. The condition action of t1 assigns to variable a the current value of the input variable, x, plus one. At the next step of execution, first the guard on transition t2 is tested, and if true, t2 is taken and the current state is state C. Otherwise, the guard on transition t3 is tested, and if true, t3 is taken and the current state is state D. If the guard on neither t2 nor t3 evaluates to true, the current state remains state B.

### 3.2 Symbolically executing a model

At a high level, symbolic execution of the model in Figure 2 means that instead of reading a specific, concrete value from the environment for the input variable x at each step of execution, a symbolic value is used instead. Whenever x is used in actions, such as transition t1, or in guards, such as transitions t2 and t3, the symbolic value is used. When a branching condition is reached, such as whether the guard on a transition is true or false, both paths are explored by forking execution and attaching a path constraint to each separate path. Along any given path of a symbolically executed program, the path constraint states what conditions must be satisfied in order for a path to be satisfiable.

Generalizing this high-level description and the sample execution, a path of symbolic execution of a Statechart model needs three pieces of information. A data structure holding this information (named SymbolicSnapshot) is shown at the top of Listing 1 and explained below.

The first item in the SymbolicSnapshot data structure, a list of active states, is also used in concrete execution. This simply tells which states in the model are active in an execution path. This list depends on symbolic values only in that the path constraints and assignments to symbolic variables must be satisfiable for this list of states (and path) to be valid.

The second item in the SymbolicSnapshot data structure is an array of symbolic values of the variables in a model. To describe a data structure for holding the variable values, consider again the model in Figure 2. We assume that initially, the model is in state A and that the value of variable a is 0. With ordinary, concrete execution, at the beginning of a step of execution, the value of the input variable x is read from the environment. However, because the model is being symbolically executed, at the beginning of each step, a symbolic value is assigned to x (this symbolic value can be denoted with a special symbol in the implementation). After a symbolic value has been assigned to the input variable x for the current step, transition t1 is tested. Because t1 has no guard condition, its condition action is performed. The condition action of t1 assigns the value of the expression $x + 1$ to the internal variable a. In order to perform this assignment, we first parse $a = x + 1$, retrieving the current symbolic value of x from the list of symbolic values. Next, we update the symbolic value of the variable a based on the assignment.

The reasons for the actions above is that, in general, assignment expressions, such as the one found on the condition action of transition t1, can be arbitrarily complex, and the values of all the variables used in the expression must be retrievable from the array of symbolic values contained by the current symbolic snapshot. However, in order to retrieve the current value of a symbolic variable, information is needed about where the variable is stored, such as an index into the array of symbolic values held by a SymbolicSnapshot.

The top-right hand portion of Figure 2 shows the example model augmented with a tree structure holding the expression in the condition action on t1. Each node in the tree in Figure 2 is an instance of the SymbolicNode data structure (Listing 1). In addition to a member variable giving its type, each SymbolicNode has an index, which is an integer index into the array of symbolic values held by the SymbolicSnapshot. When the symbolic execution engine walks the tree corresponding to an expression (in this case, $a = x + 1$), the index held by a SymbolicNode will be used to retrieve the current value of the variable from the array of SymbolicValue variables held in a SymbolicSnapshot. This allows the generated code for specific Statechart models to interact with the data held by the symbolic execution engine. The bottom part of Figure 2 describes how the values of the symbolic variables are assigned as the model is executed.

Generalizing from this example, a SymbolicValue data structure is shown in Listing 1.
Listing 1: Data structures for symbolic execution.

The third and final item in the SymbolicSnapshot data structure holds constraint information. The core of symbolic execution is what happens during branching conditions: execution is forked and an attempt is made to explore both paths. Recall that an instance of the SymbolicSnapshot contains all necessary information about a path of execution. When a branching condition is encountered, both paths can be explored in the following way. First, two copies of the current SymbolicSnapshot are made. Next, a path constraint asserting that the branch condition is true is appended to one of the SymbolicSnapshots, and execution proceeds along the “then” branch with the assumption that the branching condition is true using that SymbolicSnapshot. To the other copy of the SymbolicSnapshot, a path constraint asserting that the branch condition is false is appended, and execution proceeds along the “else” branch with the assumption that the branching condition is false using this SymbolicSnapshot.

A path constraint contains the constraints that must be satisfiable in order for a path to be valid. After execution is forked and path constraints are appended, the constraints can be given to an off-the-shelf constraint solver and checked for satisfiability. If the constraints of a path are satisfiable, symbolic execution can proceed along that path. Furthermore, valid assignments to the input variables can be used as test-inputs that are guaranteed to drive the model through the execution path represented by a SymbolicSnapshot. If the path constraints of a SymbolicSnapshot are not satisfiable, execution along that path stops.

One of the reasons SPF has interpretive overhead when analyzing models in Polyglot is that from SPF’s perspective, the Statechart model code is treated like regular Java code. This means that SPF attempts to explore both branches of every conditional statement in the semantic module code. However, there are many conditional statements found in the semantic modules that provide the implementation but are irrelevant with respect to the branching conditions of a Statechart. For Stateflow models, the branching points of interest are on the guard conditions of transitions. The custom symbolic execution engine explores both paths of only these branching points, thus avoiding the overhead of SPF.

Path constraints can be represented in various ways. The SymbolicConstraint data structure at the bottom of Listing 1 implements path constraints as a linked list of strings. When a new constraint is appended to a SymbolicSnapshot, it is linked to the previous constraints held by the SymbolicSnapshot. The simplicity of this approach facilitates multithreading (Section 3.3). However, a more complex representation, such as a tree, can also be used, especially if the constraints are preprocessed before they are given to a constraint solver.

Figure 3 shows the generated path constraints for each of the three possible symbolic execution paths that result from two steps of execution of the model in Figure 2. If in this exam-

3.3 Architecture and multithreading

Even though the custom symbolic execution engine alone significantly reduces the overhead imposed by SPF, multithreading can be used to scale symbolic execution even more. There has been work on multithreaded and distributed symbolic execution (see Section 6), but we are not aware of any such work that focuses on multithreaded symbolic execution specifically for Statechart analysis.

The overall architecture of our singletreaded approach is shown in Figure 4, which is labeled with annotations that describe the sequence of execution. The first step in the figure gives the initial symbolic snapshots to the symbolic execution engine. Next, the symbolic execution engine performs a step of execution, generating new symbolic snapshots. The newly generated symbolic snapshots may contain unsatisfiable constraints, so in order to discard the infeasible paths, the constraints for each symbolic snapshot are given to a constraint solver. Snapshots with unsatisfiable constraints are discarded, while the remainder of the snapshots are fed back to the symbolic execution engine and the process repeats to a user-specified depth.

Multithreading is implemented in our approach in the following way. After the unsatisfiable symbolic snapshots are discarded (Step 4 in Figure 4), the remaining symbolic snapshots are divided and distributed across multiple instances of the symbolic execution engine that run in separate threads. This approach is possible because the snapshots share no writable state information: the data structures for path constraints and symbolic values use linked lists that never modify previous elements. Section 5 gives benchmarks with
Our approach is to symbolically execute the Polyglot \texttt{step()} method for a particular formalism (e.g., Stateflow) for each state (and each event) in the model. Conceptually, the \texttt{step()} method takes as input a set of states and an event and then advances the Statechart to the next set of states as encoded in the Statechart. The result of symbolic execution applied to \texttt{step()} for one state is a set of path conditions together with the computed \texttt{next} states in the model and the \texttt{effects} on the global variables, expressed as functions over the symbolic inputs. We then aggregate the results for all the paths and all the states in the model, resulting in a specialized code for the \texttt{step()} method that has fewer instructions than the original code.

As an example, consider the Stateflow model in Figure 5. The result of symbolically executing the \texttt{step()} method for state G and the empty event is shown in Listing 2 (there are no effects on the global variables in this simple example).

```
Listing 2: Results of symbolic execution for state G1.

\begin{verbatim}
initial state: G
next states: D
PC: x > 0
effects: ---

next states: F
PC: !(x > 0) &\& y > 0
effects: ---
\end{verbatim}
```

Listing 3: Results of specialization.

```
Listing 3: Results of specialization.

\begin{verbatim}
\begin{tabular}{l}
\texttt{ArrayList\langle State\rangle} \texttt{sl} = \texttt{new ArrayList\langle State\rangle();}
\end{tabular}
\end{verbatim}
```

We note that our program specialization performs a more "aggressive" optimization than typical partial evaluation (see discussion in related work). Partial evaluation could be useful when we fall back on the original code, e.g. infinite loops.

5. EVALUATION

We have applied the presented techniques to the analysis of several examples that exercise different features of Statecharts, such as hierarchy, parallel states and infinite loops (i.e., loops that result in an infinite execution when executed symbolically). We report here on one particular model that contains both hierarchy and parallel states: a Statechart model containing three identical copies of the model in Figure 5, each running in parallel. For any particular step of execution, there are exactly three possible execution paths; thus, increasing the exploration depth by one increases the number of execution paths by a factor of three.

We analyzed this model to different depths using (1) the original version of Polyglot with SPF, (2) Polyglot using the specialized \texttt{step()} method that has fewer instructions than the original code, e.g. infinite loops.
the specialized code with SPF and (3) the custom symbolic execution engine with different numbers of threads. The experiments were performed on a 2.67GHz four core machine with 4GB RAM running 32-bit Windows 7 Enterprise. Our results are shown in Figure 6. As the graph shows, both the custom symbolic execution engine and SPF applied to specialized code achieve significant time savings as compared to SPF running with the original Polyglot code.

6. RELATED WORK

A work similar to ours is the symbolic execution of UML-RT models [9]. In that work, the symbolic analysis is implemented in the IBM Rational Software Architect Real Time Edition and can be used to analyze UML-RT models that use certain features of the language. In particular, [9] says that capsules with parallel behaviors and hierarchy may not be analyzable. This is in contrast to our tool, which can be used with models containing both parallel states and hierarchy. Additionally, our symbolic analysis can be distributed across multiple threads, a technique not mentioned in [9].

The parallelization of symbolic execution has been discussed in [8]. The authors describe heuristics that can be used to rank the difficulty of a symbolic execution paths according to how difficult their associated path constraints are for a solver. The symbolic execution paths can then be evenly distributed across execution threads, so that no thread gets stuck solving all of the difficult constraints. A prototype implementation is also defined for SPF. This work is complementary to our approach, and the heuristics presented in [8] could be used to rank and distribute the symbolic execution paths in our framework.

Symbolic execution has been used before for the specialization of Ada programs [4]. That approach transforms a generalized software component into a more specific and more efficient component. Symbolic execution is used to simplify each instruction in the code. In contrast we employ a more aggressive specialization, by computing directly the effects of the step() method (without generating intermediate simplified instructions), resulting in (potentially) fewer instructions to execute. Our work however applies to a Statechart modeling framework and not to a general purpose programming language.

7. CONCLUSIONS

This paper presented two approaches for improving the performance of symbolic execution with our Statechart analysis framework, Polyglot. The first was a custom, multithreaded symbolic execution engine that eliminates the overhead associated with generic symbolic execution engines. The second approach was a specialization of the generated code, resulting in equivalent code that is easier for the original symbolic execution engine to analyze.

This work can be extended in several ways. The two techniques described here could be combined, resulting in a custom symbolic execution engine that operates on specialized code. Another improvement that we plan on implementing is better solver integration. Also, the analysis of large models could be distributed across a cloud computing platform. We plan on testing such a deployment in the near future.

Finally, we note that we have done our implementations and performed our experiments in the context of Stateflow models. However, we support multiple semantics in our Polyglot original interpreter, and we see a straightforward path to support custom symbolic evaluation for UML and Rhapsody.

8. ACKNOWLEDGMENTS

The work described in this paper has been supported by NASA under Cooperative Agreement NNX09AV58A and by the NSF under grant CNS-1035715. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration or the NSF.

9. REFERENCES


