Polyglot: Systematic Analysis for Multiple Statechart Formalisms

Daniel Balasubramanian
¹, Corina S. Păsăreanu², Gábor Karsai¹, and Michael R. Lowry³

¹Vanderbilt University, ²Carnegie Mellon Silicon Valley, ³NASA Ames daniel@isis.vanderbilt.edu, corina.s.pasareanu@nasa.gov, gabor@isis.vanderbilt.edu, michael.r.lowry@nasa.gov

Abstract. Polyglot is a tool for the systematic analysis of systems integrated from components built using multiple Statechart formalisms. In Polyglot, Statechart models are translated into a common Java representation with pluggable semantics for different Statechart variants. Polyglot is tightly integrated with the Java Pathfinder verification toolset, providing analysis and test-case generation capabilities. The tool has been applied in the context of safety-critical software systems whose interacting components were modeled using multiple Statechart formalisms.

Keywords: Statecharts, symbolic execution, model checking

1 Introduction and Tool Overview

Polyglot is a unified environment in which multiple variants of Statecharts [1], a popular modeling formalism for the dynamics of reactive systems, can be executed and verified against properties. The work on Polyglot has been motivated by large programs such as human space exploration, that involve multiple systems that interact via safety-critical protocols. These systems have been designed using different Statechart formalisms to build models from which code is automatically generated. Determining the impact of using different formalisms on the reliability and safety of such model-based software has been a daunting task with little prior tool support available.

Polyglot performs the analysis of the different models (e.g. expressed in Matlab Stateflow or Rational Rhapsody) by translating them to a common intermediate representation, which is then translated into Java code that represents the "structure" of the model (see Figure 1). The semantics are provided as separate "pluggable" modules. Currently, Polyglot includes modules that implement the semantics of Matlab Stateflow, Rational Rhapsody, and UML Statemachines; the framework can be extended easily with other Statechart semantics. The Java code representing the structure of the model is combined with one of these semantic modules, resulting in an executable component. We have also developed a formal description for the various Statecharts semantics using the structural operational semantics formalism (SOS) [2] to provide confidence in our implementation. Properties of interest are expressed using specification patterns [3]

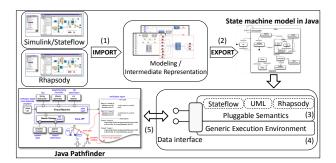


Fig. 1. The Polyglot tool.

which are automatically translated into checking code similar to observer automata [4]. The analysis is performed using Java Pathfinder (JPF) [5]. JPF is a mature open-source tool-set for the verification of Java bytecode, that incorporates model checking and powerful test-case generation (i.e. the symbolic execution tool Symbolic PathFinder – SPF [6]) and compositional verification capabilities [7]. Polyglot is written in Java and it is freely available from [8].

The clear separation between the model structure and the different semantics provides several advantages. First, it provides the basis for analyzing interacting models that operate under different semantics. This is crucial to finding interoperability and interface errors early in the design phase, since e.g. previous findings show that the majority of errors in NASA's Apollo and Skylab software were interface errors [9]. Furthermore, this approach allows users to verify whether model properties are preserved across different variants of Statecharts, ensuring that there are no misunderstandings in requirements and design development due to semantical differences. Moreover, Polyglot allows a user to understand and analyze the behavior of models across different tools in a single framework.

Verification and validation techniques exist for several individual modeling formalisms, and supporting tools offer features such as test-input generation and model checking (see below). However, existing modeling languages and analysis tools are limited to a single Statechart formalism and have limited verification capabilities. What distinguishes Polyglot from other related approaches is its extensibility both in terms of Statechart semantics that are supported (via "pluggable" semantics) and analyses that can be performed, via the extensible JPF verification framework or custom analysis.

Related Tools The analysis of Simulink/Stateflow models is supported by commercial tools such as Mathworks' *Design Verifier*, used for model checking and test case generation, and Reactive System's *Reactis* and T-VEC's *tester*, used for test generation and coverage. 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.

Polyglot is similar to the heterogeneous model analysis from [10], which is based on a common "inframodel" and a set of rules describing the semantics and interactions between multiple formalisms. The work is concerned with high-level model descriptions and it would take considerable effort to use those rules to capture the semantic details for the Statecharts that are the focus here. Also that work does not address property preservation under different semantics.

The Ptolemy environment [11] is a laboratory for experimenting with different models of computation for component based systems. Ptolemy implements polymorphic components whose behavioral semantics depend on an "execution engine" ("director" in Ptolemy) similar to our "pluggable semantics". Our work addresses different Statechart variants and formal semantics with particular focus on model checking and systematic test case generation, while Ptolemy's goal is simulation.

The parametric semantics from [12–14] provide powerful semantic frameworks for many Statecharts variants as well as process algebras. While quite flexible, they can not fully capture the behavior of any of the three notations considered here (see [15] for details).

2 Design Choices and Extensions

Design Choices 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. Java has a precise, clear semantics, well-understood by many, so implementing a concise simple execution engine for the Statechart variants (that is actually readable) is a good, pragmatic approach to defining semantics. 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 to leverage the model checking and symbolic execution capabilities from JPF for systematic analysis, automated test case generation (with SPF) and coverage measuring.

We also note that the Statechart variants 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, would be 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 action-language features into a similar concept in Java.

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 driving the execution non-deterministically or symbolically.

Extensions The integration of Polyglot with JPF enables us to take advantage of the optimized analysis techniques that are already provided by JPF. To further improve the performance of Statechart analysis in Polyglot, we have experimented with two techniques [16]. The first is a multithreaded custom symbolic execution engine for Polyglot, while the second technique is the application of partial evaluation to optimize the generated Polyglot code with respect to partic-

4 Balasubramanian et. al

ular models and semantics. We note that the design of Polyglot, which decouples the semantic modules from the "structure" of a Statechart model, lends itself well to a multithreaded implementation.

Polyglot can be used as described above to execute and analyze both individual models and also systems with a simple communication that matches Statechart semantics (i.e. event broadcast). This mechanism is insufficient for components that execute in parallel and communicate asynchronously. The problem could be addressed by modeling the communication protocol itself as another Statechart and composing it with the other models. However this may be inefficient, as the protocols can be very large. We have therefore explored extending Polyglot with features not inherent to the basic Statecharts paradigm. These include a connector mechanism for communication and a scheduling framework for sequencing the execution of individual components [17].

Polyglot comes with a library of connectors modeling lossless FIFO communication and ARINC-653 ¹ ports. Instead of reading data from or sending data directly to another component, data is read from or written to a connector. Other communicating mechanisms, such as lossy communication and non-FIFO message delivery, can be easily incorporated. The scheduler is responsible for ordering the component execution and for invoking the property checking. We have developed a generic scheduler that can be instantiated with different scheduling mechanisms, e.g. non-deterministic, priority-based, calendar-based, etc. By default, Polyglot uses a non-deterministic scheduler. Currently, it is the responsibility of the user to manually link the components via the connector and scheduling mechanism. We intend to automate the process using the Generic Modeling Environment (GME) [18], a graphical tool that already supports our intermediate representation and in which we can describe a system's architecture and automatically generate the code for connector and scheduler instances.

3 Tool Usage

Polyglot has been applied to medium-sized models of flight software, including an example modeling a component from NASA/JPL's Mars Exploration Rovers (MER) [15]. The MER software consists of a Resource Arbiter and several user components, serving specific applications, such as imaging, controlling the robot arm, communicating with earth, and driving. The arbiter moderates access to shared resources, preventing potential conflicts between resource requests and enforcing priorities; e.g., a communication session with Earth can not be started while the rover is driving. Each user has 2 pseudostates, 4 atomic states, 1 compound state and 9 transitions (259 LOC in the Java representation), while the arbiter has 33 pseudostates, 15 atomic states, 2 orthogonal states and 58 transitions (1788 LOC). Polyglot was used for checking safety properties and generating test cases for this model, where the semantics of User 1 was changed from Stateflow into UML and Rhapsody. Table 1 shows the results for analyzing

¹ Avionics Application Standard Software Interface, Aeronautical Radio, Inc.

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

Table 1. Experimental results.

the models with increased number of time steps, corresponding to sequences of sizes $4,\,5$ and 6.

The property holds for the Stateflow models, but it fails when we change the semantics of one user to UML or Rhapsody. This is due to a semantic difference between UML and Stateflow (outer transitions have higher priority over inner transitions in Stateflow, but have lower priority in UML and Rhapsody). This semantic difference is also reflected in the different number of test cases. Note that the results for UML and Rhapsody are practically identical (since their semantic differences are not exposed by the analyzed models).

The feedback produced at the Java-level has the form of test sequences that have been used as inputs to drive the simulation of the models in the original modeling environments. The generated test sequences can also be used for testing the code that is generated from the models.

Polyglot has been used also to analyze models representing the interaction between the Ares launch vehicle and the Orion Crew Exploration Vehicle [17]. The Ares-Orion communication during abort was formulated as a property derived from the official flight software design documents and the software requirements specification available for Ares I. The analysis confirmed problems suspected by the engineer who developed the model, who had already submitted a request for a change to the Ares I design document. Since then, the design has changed to reduce the command echo dependency because of a bit-rate limitation. The effects of that change have not yet been investigated, but our tool can help answer this for the future.

4 Conclusion

We have described Polyglot, a tool for the systematic analysis of model-based software written with multiple Statechart formalisms. The tool has been applied to the analysis of safety-critical systems whose interacting components were modeled using multiple Statechart formalisms. We plan to further expand and robustify the tool and use it for the analysis of the ground system in the GOES-R project [19]. We also plan to explore the compositional techniques from JPF [7]

for the component-based analysis of models in Polyglot. As model-driven development is increasingly used in a diverse way for the design and implementation of safety and mission critical systems, we believe that our tool will provide a key capability for the verification and validation of such software.

Acknowledgments This work has been supported in part by NASA under Cooperative Agreement NNX09AV58A. The authors would also like to thank Michael Whalen for valuable discussions and feedback.

References

- Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3) (June 1987)
- 2. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Comp. Sci. Dept., Aarhus University, Denmark (1981)
- 3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE. (1999)
- Balasubramanian, D., Pap, G., Nine, H., Karsai, G., Lowry, M.R., Pasareanu, C.S., Pressburger, T.: Rapid property specification and checking for model-based formalisms. In: International Symposium on Rapid System Prototyping. (2011)
- $5.\,$: Java pathfinder. http://babelfish.arc.nasa.gov/trac/jpf
- Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proceedings of ASE. (2010) 179–180
- Giannakopoulou, D., Pasareanu, C.S.: Interface generation and compositional verification in javapathfinder. In: FASE. (2009)
- 8. : Polyglot. https://wiki.isis.vanderbilt.edu/MICTES/index.php/Publication
- 9. Hamilton, M.: 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)
- Pezzè, M., Young, M.: Constructing multi-formalism state-space analysis tools: Using rules to specify dynamic semantics of models. In: ICSE. (1997)
- 11. Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y.: Taming heterogeneity the ptolemy approach. Proc. of IEEE **91**(1) (2003)
- 12. Esmaeilsabzali, S., Day, N.A., Atlee, J.M., Niu, J.: Big-step semantics. Technical Report CS-2009-05, David R. Chariton School of Computer Science, Univ. of Waterloo, Ontario, Canada N2l 3G1 (2009)
- 13. Esmaeilsabzali, S., Day, N.A.: Prescriptive semantics for big-step modelling languages. In: 13th Int. Conf. Fundamentals of Softw. Eng. (2010)
- Niu, J., Atlee, J.M., Day, N.A.: Template semantics for model-based notations. IEEE Trans. Software Eng. 29(10) (2003) 866–882
- Balasubramanian, D., Pasareanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple statechart formalisms. In: ISSTA. (2011)
- Balasubramanian, D., Pasareanu, C.S., Karsai, G., Lowry, M.R., Whalen, M.W.: Improving symbolic execution for statechart formalisms. In: MODEVVA. (2012)
- Balasubramanian, D., Pasareanu, C.S., Biatek, J., Pressburger, T., Karsai, G., Lowry, M.R., Whalen, M.W.: Integrating statechart components in polyglot. In: NFM. (2012)
- Dubey, A., Karsai, G., Mahadevan, N.: A component model for hard real-time systems: Ccm with arinc-653. Softw., Pract. Exper. 41(12) (2011) 1517–1550
- 19. : Goes-r. www.goes-r.gov

Appendix: Tool Demonstration

The demonstration will focus on the following capabilities of Polyglot.

- 1. Simulating models with different semantic variants.
- 2. Defining properties with our pattern based specification system.
- 3. Executing models symbolically with SPF (the symbolic execution mode of JPF) to identify property violations.
- 4. Generating test cases with symbolic execution, using the SPF tool.

The following sections describe each of these pieces in more detail.

Simulating models with different semantics

The demonstration will begin with a short example that highlights Polyglot's most distinguishing feature: quickly executing the same Statechart model using different semantics. The model we will use is shown in Figure 2. This model is interesting because the occurrence of event "e" results in a different state configuration for Rhapsody, UML and Stateflow semantics. Using Polyglot to demonstrate these differences involves the following steps.

- 1. Defining the model.
- 2. Translating the model to Java.
- 3. Running the model with different semantic variants.

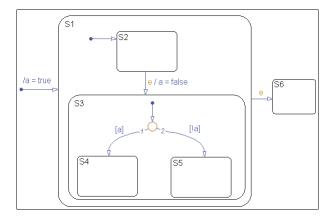


Fig. 2. Simple model to show semantic differences.

We will perform the first step using Matlab Stateflow, although both the Rational Rhapsody tool and our intermediate language can also be used instead. For

Stateflow, we use a mature translation tool, MDL2MGA, to generate the intermediate language. MDL2MGA and the intermediate language have already been used by several other large projects (with NASA and DARPA), and we reuse them in Polyglot. MDL2MGA is described on Page 7 of the documentation included with the Polyglot distribution. Note that our intermediate representation (IR) is supported by the graphical environment GME, so that users can draw the models using GME in the IR and then generate Java code from that.

The distribution also includes a tool called *Rhapsody2MGA* that we built specifically for Polyglot that translates from Rational Rhapsody models into our intermediate language. Instructions for using *Rhapsody2MGA*, along with a sample Rhapsody model, are available in the "samples/rhapsodyex1" directory in the distribution. Limitations of the *Rhapsody2MGA* translator are discussed on page 26 of the Polyglot documentation.

Note that there is no such translation tool for UML. While there are many tools for building UML models, the degree to which most of these tools conform to the semantics described by the official OMG specification for State Machine Diagrams is unclear. We made a strong effort to understand these semantics and describe them using the SOS formalism, and then implement them in Polyglot. To use the UML semantics, users can define models in Stateflow, Rhapsody or the IR, and then translate the models to Java code and run them with the UML semantic module.

The third step above, running the model with different semantic variants, is performed by compiling the generated Java code and running it with the semantic modules. Changing the semantic module with which the generated Java code runs involves changing a command line parameter. We will run the model with three different semantics (UML, Rhapsody and Stateflow) and compare the final state configurations for each.

This example is included in the Polyglot distribution in the "samples/semantic differences" folder.

Defining properties

The next part of the demonstration will show how property specification and checking are integrated into Polyglot. This part will use a Statechart model for a cruise controller, shown in Figure 3. After describing the model, this part will involve the following steps.

- 1. Defining two temporal properties.
- 2. Translating the model and properties into Java code.
- 3. Running the Java code in simulation mode.

The two temporal properties will state the following (the first is satisfied by the model but the second is not).

- The "CC_engaged" state must be active before the "CC_paused" state.

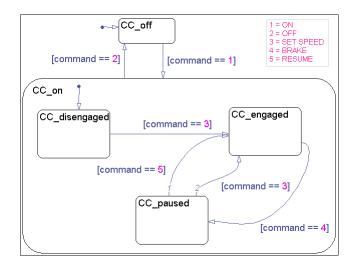


Fig. 3. Cruise controller model.

- The model should not be in the "CC_on" state at the end of a step in which the input command is "set speed".

The first property is an example of the *precedence* pattern: a state or condition must be preceded by a certain combination of states and conditions. The second property is an example of the *absence* pattern: a certain state or condition should never happen during some scope of execution.

We have built a graphical extension for the Matlab Stateflow environment that allows properties to be entered into dialog boxes using a pattern based system. Figure 4 shows a screenshot of these dialog boxes with the two properties above. Once the properties have been entered, they are automatically translated into Java code that is integrated with Polyglot.

After the model and properties are translated into Java code, we will manually simulate the model and provide an input sequence that leads to a violation of the second property. This will demonstrate how Polyglot can check properties in simulation mode.

This example is included in the Polyglot distribution in the "samples/cruisecontroller" folder.

Automatically finding property violations

This part of the demonstration will build on the previous step, in which the property violation was discovered by manually entering inputs. In this phase, we will use the same model and unsatisfied property and show how SPF can automatically find input sequences that violate the property.

We will begin by showing how to configure SPF to analyze our model. This consists of writing an SPF configuration file and specifying necessary parameters.

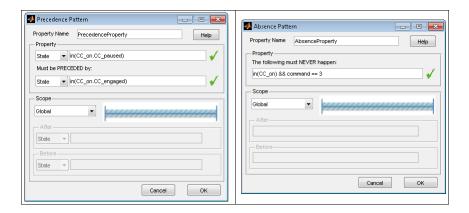


Fig. 4. Screenshot of the property specification system.

After this, SPF will automatically explore different paths through the model and report when it finds a path leading to a property violation.

Test-case generation

This part of the demonstration will show how Polyglot can use SPF to find input sequences to reach states when a model has more complex arithmetic constraints and actions. The Statechart model for this part is shown in Figure 5.

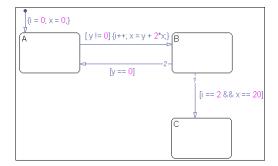


Fig. 5. Model with more complex constraints and actions.

The goal is to find an input sequence that reaches state C. The guard on the transition to state C says that the value of internal variable i is 2 and the value of the internal variable x is 20. However, at first glance, it is not obvious what sequence of values to the input variable y will lead to these conditions being satisfied.

We will first show how to configure SPF to explore this model. Next, we will execute this model symbolically using SPF. The relevant output is at the top of listing 1.1. Line 2 is generated by SPF and shows a sequence of method calls that can be given to the model in Polyglot to drive it to state C. Lines 5 through 14 show the state sequence and values for the variable x when this test-input is provided to the model.

This example is included in the Polyglot distribution in the "samples/symbolicex1" folder.

```
// Output from SPF
   [\operatorname{setData}(1), \operatorname{setData}(0), \operatorname{setData}(18), \operatorname{setData}(-4), \operatorname{setData}(-1)]
   // Inputting this test-sequence to Polyglot
   Initial condition:
   current state = A, value of x = 0
   After setData(1):
   current state = B, value of x = 1
   After setData(0):
10
   current state = A, value of x = 1
11
   After setData(18):
   current state = B, value of x = 20
12
13 After set Data (-4):
14 current state = C, value of x = 20
```

Listing 1.1. Data structure for a symbolic execution path.

Obtaining Polyglot

The Polyglot distribution is available for download at http://balasub.com/polyglot/. All of the demo models described in this appendix are included as samples in the distribution. The website also includes two sample videos showing Polyglot.

The distribution has the following structure; we comment only on the parts relevant to this appendix. Full details are found in the readme.txt file in the top-level directory.

- /bin: The MDL2MGA and Rhapsody2MGA translators, the Java code generator and a .jar file of the Polyglot libraries.
- /doc: The main documentation (pgdesign.pdf).
- /etc: The .xsd for our intermediate language.
- /samples: Includes all of the samples described in this appendix along with additional examples.
- /src: The MATLAB files needed for defining temporal properties inside MATLAB, and the Polyglot source code.

The prerequisites for running Polyglot are Windows 32-bit (for running the code generation tools), version 6 or later of the Java JDK and JPF/SPF.

12 Balasubramanian et. al

If time permits, we will show an overview of the Polyglot code, and we will demonstrate the use of connectors and the non-deterministic scheduler. In addition to the capabilities described here, we have implemented a multithreaded, custom symbolic execution engine for Polyglot, which can improve the execution time of the symbolic analysis by more than an order of magnitude. Time permitting, we would also like to showcase this as a highlight of how Polyglot's original design that uses pluggable semantics allows powerful extensions to be built.