Polyglot: Systematic Analysis for Multiple Statechart Formalisms*

Daniel Balasubramanian[1], Corina S. Păsăreanu[2], Gábor Karsai[1], and Michael R. Lowry[1]

[1] Vanderbilt University
[2] Carnegie Mellon Silicon Valley
[3] NASA Ames
{daniel,gabor}@isis.vanderbilt.edu,
{corina.s.pasareanu,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 tool-set, 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

* The rights of this work are transferred to the extent transferable according to title 17 U.S.C. 105.

N. Piterman and S. Smolka (Eds.): TACAS 2013, LNCS 7795, pp. 523–529, 2013.

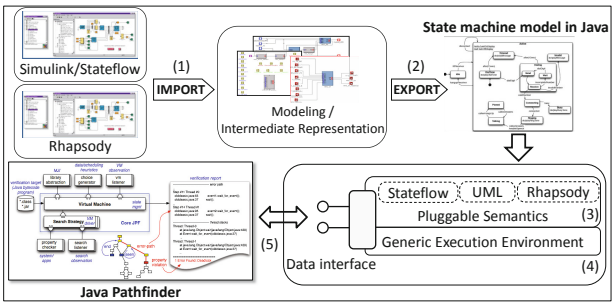

Fig. 1. The Polyglot tool

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] 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 particular 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. 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

Table 1. Experimental results

Semantics, Seq. size	Total # Test Cases	Property	Memory, Time
U1 Stateflow, 4	125	true	20 MB, 43 s
U1 Stateflow, 5	412	true	22 MB, 2 m 04 s
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	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

Rhapsody. Table 1 shows the results for analyzing 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 and Tom Pressburger for valuable discussions and feedback.

References

1. 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)
4. 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`
6. Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proceedings of ASE, pp. 179–180 (2010)
7. Giannakopoulou, D., Păsăreanu, C.S.: Interface Generation and Compositional Verification in JavaPathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
8. Polyglot, `https://wiki.isis.vanderbilt.edu/MICTES/index.php/Publications`
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)
10. 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: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 158–172. Springer, Heidelberg (2010)
14. Niu, J., Atlee, J.M., Day, N.A.: Template semantics for model-based notations. IEEE Trans. Software Eng. 29(10), 866–882 (2003)
15. Balasubramanian, D., Pasareanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple statechart formalisms. In: ISSTA (2011)

16. Balasubramanian, D., Pasareanu, C.S., Karsai, G., Lowry, M.R., Whalen, M.W.: Improving symbolic execution for statechart formalisms. In: MODEVVA (2012)
17. Balasubramanian, D., Păsăreanu, C.S., Biatek, J., Pressburger, T., Karsai, G., Lowry, M., Whalen, M.W.: Integrating Statechart Components in Polyglot. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 267–272. Springer, Heidelberg (2012)
18. Dubey, A., Karsai, G., Mahadevan, N.: A component model for hard real-time systems: Ccm with arinc-653. Softw., Pract. Exper. 41(12), 1517–1550 (2011)
19. Goes-r, `http://www.goes-r.gov`