Integrating Statechart Components in Polyglot

Daniel Balasubramanian[1], Corina S. Păsăreanu[2], Jason Biatek[3], Thomas Pressburger[4], Gabor Karsai[1], Michael Lowry[4], and Michael W. Whalen[3]

[1] Vanderbilt University/ISIS, 1025 16th Ave S, Nashville, TN 37212
[2] Carnegie Mellon Silicon Valley, NASA Ames, M/S 269-2, Moffett Field CA 94035
[3] University of Minnesota, Dept. of Comp. Sci. and Eng., Minneapolis, MN 55455
[4] NASA Ames Research Center, M/S 269-2, Moffett Field, CA 94035

Abstract. Statecharts is a model-based formalism for simulating and analyzing reactive systems. In our previous work, we developed Polyglot, a unified framework for analyzing different semantic variants of Statechart models. However, for systems containing communicating, asynchronous components deployed on a distributed platform, additional features not inherent to the basic Statecharts paradigm are needed. These include a connector mechanism for communication, a scheduling framework for sequencing the execution of individual components, and a method for specifying verification properties spanning multiple components. This paper describes the addition of these features to Polyglot, along with an example NASA case study using these new features. Furthermore, the paper describes on-going work on modeling Plexil execution plans with Polyglot, which enables the study of interaction issues for future manned and unmanned missions.

Keywords: Statecharts, analysis, modeling, testing

1 Introduction and Motivation

This paper reports on an on-going project at NASA Ames, whose goal is to develop early, *design-level* automated techniques for error detection in the flight control software developed for the next generation of manned and unmanned space missions. The Ares-Orion abort scenario for the Constellation program was an original motivating example for this work and is also used in this paper to illustrate the technical capabilities of integrating different Statechart components in our modeling and analysis framework.

During the Constellation Program, NASA was determined to provide a last-chance option for astronaut survival if the Ares launch vehicle exploded during launch – as did the rocket booster for the Space Shuttle Challenger in 1986 – and therefore spent significant resources on a launch abort system. The driving requirement was to provide the Orion crew capsule with a powerful abort rocket capable of rapidly pulling the capsule away in case of an explosion. The Ares launch vehicle on-board fault diagnostics would interact with the Orion spacecraft'scontrol system to detect an emerging hazard and execute either a crew-initiated or automated firing of the launch abort rocket. Achieving the

rapid control capability for a launch abort became a major design driver for the Orion software architecture, the Ares software architecture, and also the interface between Ares and Orion.

The interface requirements between Ares and Orion were defined in an English-language Interface Control Document that included communication and control specifications to be implemented by the Ares and Orion flight software. Both Ares and Orion had adopted model-based software design methods. However, due to cultural reasons and the technical capabilities of different tools, a multitude of modeling formalisms were adopted: Enterprise Architect (UML 2.0) for Ares, Mathworks Simulink/Stateflow for math-intensive functions on Orion, and Rhapsody for the overall software framework for Orion. The Statechart control component for these different modeling formalisms each has different execution semantics. This makes performing conventional formal methods analysis of interacting systems developed with these different modeling formalisms difficult.

In previous work [2] we developed Polyglot, a framework for modeling and analysis of software using different Statechart formalisms. Polyglot uses a common intermediate representation with customizable Statechart semantics and leverages existing verification and test case generation technologies developed at Ames [1, 4]. However, to study integration issues between asynchronous components described using different modeling formalisms, as in the Ares-Orion case study, additional features need to be added to Polyglot. These include a connector mechanism for modeling communication, an execution scheduling framework and a method for specifying verification properties that span multiple components. This paper describes the addition of these features to Polyglot, along with an analysis of the Ares-Orion abort scenario using these new features. We also describe on-going work on modeling Plexil [3] execution plans with Polyglot, which enables the study of interaction issues for future manned and unmanned (robotic) missions. Although we make our presentation in the context of a particular NASA project, we believe that our work should be relevant to other complex, safety critical model-based software that is built from multiple components modeled with different Statechart formalisms.

2 Integrating Statechart Components in Polyglot

Due to space constraints, we present here only a brief review of the typical usage of Polyglot; for a detailed description, see [2]. The basic Polyglot framework is used in the following way. First, the structure of the Statechart model (expressed in Matlab Stateflow, or Rational Rhapsody) is translated into a common intermediate representation (IR). The IR is then translated into Java code that represents the structure of the model. Only the structure of a model is translated because the semantics are provided as "pluggable" modules. Currently, modules implementing the semantics of Matlab Stateflow, Rational Rhapsody, and UML Statemachines are provided. The Java code representing the structure of the model is combined with one of these semantic modules, resulting in an executable component. Analysis can be performed using Symbolic Pathfinder

(SPF), the symbolic execution module of Java Pathfinder (JPF), which provides test-case generation and reachability analysis.

Polyglot can be used as described above to execute and analyze both individual models and also systems with simple communication between multiple models where the communication semantics matches that of Statecharts (i.e. event broadcast). However, large systems often contain components that execute in parallel and communicate asynchronously, and the basic Statecharts formalism does not provide a way to model either asynchrony or non-trivial communication between components. The remainder of this section gives a high-level overview of the connector and scheduling frameworks that were added to Polyglot for modeling communicating, asynchronous components, and also describes how properties spanning such components can be specified and checked.

Connectors The connector framework provides a generic way for components to communicate. From a component's point of view, a connector is simply a source (destination, resp.) of inputs (outputs). Instead of reading data from or sending data directly to another component, data is read from or written to a connector. The connector is responsible for determining both how data is queued when it arrives and the order in which messages are delivered when data is read.

Our basic implementation of connectors exposes two methods, *recvFrom* and *sendTo*, which components call to receive data from or write data to the connector. Sending data to a connector is non-blocking, but attempting to read from a connector that has no available data will block the calling component. This block happens on the level of the scheduling framework, so that upon being blocked, the component returns control to the scheduler. A component becomes unblocked, and thus eligible to be run by the scheduler, when another component sends data to it through a connector. The connector that we used in the experiment in Section 4 was lossless and messages were delivered in FIFO order. Another connector that we developed implements ARINC-653 [5] ports. Our intention is to develop an extensive library of connectors, modeling different communication mechanisms, including lossy communication and non-FIFO message delivery.

Scheduler The scheduling framework is responsible for determining the order of component execution and 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. The default non-deterministic scheduler implementation works in the following way. First, each Statechart component is registered with the scheduler and marked as "ready" for execution. The scheduler is then run, and upon each step of its execution, it non-deterministically runs a single step of a component that is either "ready", meaning it previously ran without blocking and is ready again, or "unblocked", meaning that the component was blocked during its previous execution step (when trying to read data from an empty connector, for instance), but has since become unblocked by the occurrence of some external event (such as having data sent to it through a connector). Unblocked components are invoked so that they can continue executing at the point at which they last became blocked,

[5] Avionics Application Standard Software Interface, Aeronautical Radio, Inc.

if desired. After the selected component finishes a step of execution, properties (described below) are checked.

Additionally, the scheduler is implemented such that if JPF or SPF are being used, all of the feasible paths with respect to which eligible (i.e., ready or unblocked) component is chosen to run are explored. This allows JPF to explore all possible valid orderings of component execution.

Properties Checking properties that span multiple components (i.e., the property involves the state configuration of more than one Statechart model) involves two main tasks. The first is specifying the property. The second is deciding when to check for property satisfaction. We specify properties using observer automata defined as Statechart models because it allows us to leverage the existing framework for translating high-level automata descriptions into Java code that can be executed directly by Polyglot. If the individual components are modeled in different tools, then the property can still be modeled as a Statechart in any one of those tools and then translated into Java.

The relevant state variables and state configuration of the components being observed are modeled as inputs to the observer automata. However, in the generated Java code, the values of these inputs are set directly by the observer automata by using references to the individual components. The observers can look directly inside the components being monitored thus eliminating the need for the Statechart components to pass any messages to the observer automata.

All properties are checked by the scheduler after each step of execution by a component, i.e. after each step of the state machine that implements the component. Because the properties are defined as observer automata using Statechart models, they are translated into Java code and executed like normal Statechart components (with the only difference being that the observers set the values of their inputs at each step by looking directly inside the monitored components). Properties that are not satisfied trigger an exception, which can be caught by SPF. The sequence of input values leading to the property violation is also reported by SPF.

3 Integrating Plexil

To further extend the reach of Polyglot, we have recently added support for Plexil [3], a PLan EXecution Language that is being used in developing various mission software for e.g., the K10 Rover [5] and human habitat automation. Plexil is based on hierarchical state machines, but unlike the other notations in Polyglot, the state machines in Plexil are implicit in the definitions of *nodes*, which describe the computational activities for executing a plan. In addition, Plexil has several language features useful for planning that are not included in the other notations, such as an extended type system in which all variables can take on the value "unknown", and a variety of different node types that have template behaviors for several activities commonly required for plan execution.

As it is likely that Plexil plans will be integrated into complex mission software involving Rhapsody, Simulink, and UML Statecharts, we want Polyglot to

have the capability of simultaneously analyzing models in all of these notations. To that end, we have added support for translating Plexil plans into Polyglot state machines whose execution model matches the Rhapsody semantics. The most significant aspect of the translation is to make explicit the implicit state machines in the Plexil plan, and to add support for the extended type system used in Plexil plans. We have added the type extensions through a Java class library that in turn is loaded into Polyglot for interpretation in JPF. There are several benefits of translating into Rhapsody besides the obvious integration into Polyglot. First, it is possible to visualize the state machines involved in Plexil nodes using the IBM Rhapsody tool suite. Second, it is possible to use the tool suite to generate code for Plexil plans.

The translation is schematic in the structure of the Plexil plan and is based on the operational semantics of Plexil [3]. However, it is currently not well-optimized, and the Rhapsody semantics impose a certain amount of inefficiency on top of the analysis due to some mismatches between the Rhapsody and Plexil conception of state machines. In the future, we are planning to perform two additional steps with respect to Plexil. First, unlike the other Statecharts notations, there is a single semantics for the Plexil Statecharts. Therefore, there is not the same utility to "swapping out" of multiple Statecharts semantics for Plexil plans. We plan first to create a better optimized translation into Polyglot in which we create a custom interpreter for Plexil plans to better match the Plexil state machine semantics. In addition, we are examining a direct-to-Java code generation option for Plexil plans as it allows still more efficient analysis.

4 Experience

The extensions to Polyglot presented in this paper were applied to models representing the interaction between the Ares launch vehicle and the Orion Crew Exploration Vehicle described in Section 1. An Ares engineer modeled both Ares and Orion in Stateflow. The Ares Stateflow model consists of six concurrent regions, each containing a state machine, while the Orion Stateflow model consists of five concurrent regions, each containing its own state machine. The inputs for this model consist of ten different boolean signals. We analyzed the component interactions using the non-deterministic scheduler described in Section 2.

We analyzed the Ares-Orion communication during abort by formulating a property derived from the official flight software design documents and the software requirements specification available for Ares I. The property states that: *"Ares aborts only if Orion initiates abort or crew commands automatic abort."*

We formulated the property as an observer automaton (as described in Section 2) which is advanced whenever the Ares or Orion components execute one step through their associated state machines. Using Symbolic Pathfinder to check this property resulted in a property violation in a 3 step sequence leading to the error. The generated test sequence revealed that Ares could also abort when there is loss of communication. Based on this analysis, we formulated a new property that, when analyzed with SPF, holds on the system.

Our 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. Even though NASA's manned space flight program has moved beyond project Constellation, the same cultural and technical factors that led to multiple modeling formalisms used in interacting safety-critical systems will persist for future missions. Our framework provides automated formal methods tools for the analysis of interactive components modeled with multiple Statechart formalisms, not only Stateflow as we discussed for this case study, as well as robotic plan execution represented by Plexil plans. This will be a key capability for verification and validation of future manned and unmanned missions.

We have implemented the component framework presented here in Java, and based on our profiling results with the Ares-Orion scenario and also with other examples, we made improvements to the performance of Polyglot when used with SPF. Our original analysis using the non-optimized version of Polyglot took a total of 4m, 15s. The optimized version of Polyglot took 2m, 2s, over 50% less time compared to the original version.

5 Conclusions

We have presented a high-level overview of three extensions to Polyglot that allow systems with communicating, asynchronous components to be modeled and analyzed. These extensions are a connector framework for modeling communication, a scheduling framework for sequencing component execution and a method for specifying properties spanning multiple, asynchronous components. A NASA case study using these extensions was described, as well as our on-going work to support the analysis of Plexil plans in Polyglot.

We continue to work on the Plexil integration and to apply our framework to the analysis of interacting software components developed for human and robotic missions. We also plan to investigate program specialization via symbolic execution to increase the speed of our analysis. This involves using SPF to specialize the Polyglot semantic modules with respect to particular Statechart models.

The Polyglot framework is available in open source form, and we plan to make the scheduling and connector framework available as well.

References

1. Java Pathfinder tool-set. http://babelfish.arc.nasa.gov/trac/jpf, 2011.
2. D. Balasubramanian, C. S. Păsăreanu, M. W. Whalen, G. Karsai, and M. R. Lowry. Polyglot: modeling and analysis for multiple statechart formalisms. In *ISSTA*, 2011.
3. G. Dowek, C. Muñoz, and C. S. Păsăreanu. A small-step semantics of PLEXIL. Technical Report 2008-11, National Institute of Aerospace, Hampton, VA, 2008.
4. C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, pages 15–26, 2008.
5. V. Verma, V. Baskaran, H. Utz, R. Harris, and C. Fry. Demonstration of Robust Execution on a NASA Lunar Rover Testbed. In *iSAIRAS*, 2008.