Reachability Verification of Rhapsody Statecharts

Kumar Madhukar
TRDDC, TCS
Pune, India
kumar.madhukar@tcs.com

Ravindra Metta
TRDDC, TCS
Pune - 411013, India
r.metta@tcs.com

Priyanka Singh
TRDDC, TCS
Pune - 411013, India
priyanka38.s@@tcs.com

R. Venkatesh
TRDDC, TCS
Pune - 411013, India
r.venky@tcs.com

***Abstract*—We present the first fully automated approach for the verification of Rhapsody statecharts. IBM's Rhapsody framework is widely used in the automotive industry to model embedded reactive systems. The reactive behavior is specified using Rhapsody's statechart formalism and controls the entire system. Hence, it is crucial to ensure the safety properties of statecharts. Therefore, we constructed a model-checking based approach to verify state reachability, a fundamental safety property, of Rhapsody statecharts. We implemented it in a prototype tool using the model checkers CBMC and SPIN. This tool successfully verified simple models, but failed to scale to industry models due to the sheer complexity of the models. We then designed and implemented a simulation based approach. This successfully verified the simple models and the industry models, and found a crucial bug in one of the industry models. In this paper, we share both our model-checking and simulation approaches, their implementation details and the experimental results.**

I. Introduction

Software embedded in automotive systems needs to satisfy critical safety and security requirements. Many countries and companies mandate their automotive software to adhere to stringent standards such as DO-178B and ISO26262. An automotive software system typically consists of many subsystems ranging from infotainment to engine control. All these subsystems need to adhere to the strict safety requirements, as even seemingly innocuous subsystems may impact the safety and security of the entire system. For example, many automotive infotainment systems provide GPS, telephone and multimedia facilities. The GPS assistance is critical since the GPS directions need to be displayed in time, and not afterwards, and should be right on top of any other video being played on the screen. Any bugs here may lead to improper GPS assistance, which in turn may lead to an accident. Similarly, in case of an emergency such as a crash, the car audio is required to be muted before an emergency phone call has to be made to an appropriate helpline.

Modeling frameworks are used for the specification of requirements of such embedded systems. They make modeling easier and promote early verification and validation of requirements through formal analysis. Rhapsody [6] is one of the popular modeling frameworks used in the automotive industry. IBM currently owns this tool. Rhapsody supports entire UML and allows code to be embedded in the models. The language of the embedded code can be chosen from C, C++ and Java. Once a language is chosen for the embedded code, Rhapsody's code generator can generate code for that model only in that language. In Rhapsody, statecharts [5] are used to specify the reactive behavior of the system being modeled. These statecharts are basically object oriented, concurrent, hierarchical state machines (see Section II-A).

The automotive Rhapsody models, which we deal with, use the Rhapsody statecharts with embedded C++ code. In order to verify these statecharts for state reachability, one needs to simultaneously take into account (a) the semantics of the Rhapsody statecharts, (b) the semantics of the rest of the UML constructs in the model and the interactions thereof, and (c) the embedded C++ code. Therefore, in order to verify such models, one needs to focus both on high level modeling formalisms and the low-level code details at the same time. This makes verification challenging and hardly scalable to industrial models, as we show later in this paper.

Model checkers are the de facto tools of choice for the verification of state machines and code [14]. There are many useful verification tools for both models and code, built using different model checkers ([13], [16]). All of them focus either exclusively on high level models or exclusively on code, but not on both. Further, these tools also focus on particular aspects of the system under verification and need lot of fine tuning to be successful on industrial systems [15]. To fully automatically verify Rhapsody statecharts, one needs to verify the model and code level details together. It is probably for this reason that none of the existing model checkers support automated verification of Rhapsody models.

In this paper we propose a new approach for verification of these statecharts. In particular, we target the verification of the state reachability property. A state machine is said to satisfy this property if each state in it can be reached from its initial state. Our first approach is based on model checking. For model checking, we need a common representation for a given Rhapsody model and the C++ code embedded in it. Since it is hard to lift C++ code to model level, we translate the model to C++ code with just enough details to faithfully capture the Rhapsody semantics. For this, we first use the Rhapsody's built-in C++ code generator to translate the Rhapsody models into C++ code. We then perform a static analysis of the C++ code to gather certain information about the statecharts. This is the only way to gather the information as Rhapsody framework does not provide a programmatic access to the models. We then eliminate those implementation details of the generated code that are irrelevant for verification and lift the code closer to the models. This leaves us with a modified C++ code that represents the original model in succinct and sufficient detail to enable verification.

There are neither academic nor commercial model checkers that can verify industrial C++ code. However, there are excellent model checkers for C such as CBMC [12]. Therefore, we translate the modified C++ code into C using the EDG's C++ to

978-0-7695-4993-4/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSTW.2013.73

C translator [19]. Next, to take care of the execution semantics of Rhapsody models, we generate C code that mimics the execution environment of Rhapsody (see Section II-A). At the end of all this, we have C code that faithfully represents the given Rhapsody model. We automated this entire process in a prototype tool that produces the final C code from a given Rhapsody model and also generates two drivers: one each for CBMC and SPIN. Now the state reachability verification on the original model translates to checking if a state variable takes on particular values in the final C code. We accordingly invoke CBMC and SPIN drivers to verify desired state reachability. In our initial experiments, this approach did not scale to even simple models. Then we did several optimizations to represent the Rhapsody environment and semantics much more succinctly in C. With these optimization, both CBMC and SPIN successfully verified the simple models.

Next, we experimented the prototype on five client supplied Rhapsody models that belong to an industrial automotive infotainment system. SPIN scaled for one of the models and did not scale for the others. CBMC did not scale for any of the models. The failure of the model checkers to scale was due to the sheer complexity of the models and the embedded C++ code. We next designed a guided simulation approach that generates a sequence of environmental inputs targeted at driving the models to desired states. We coded this approach as a part of our prototype. With this approach, our tool scaled to all the five models and demonstrated the reachability to all but one of the states. Upon manual inspection, we found that this state indeed cannot be reached as the statechart waits forever for a response that is ignored by the system due to a design error. When we showed this to the developer, he acknowledged the error and fixed the model. After this fix, our tool demonstrated reachability of this state too.

To the best of our knowledge, our model checking and simulation approaches are the first ever completely automated approaches for the verification of Rhapsody statecharts. In the remainder of this paper we detail our approaches, implementation and the experimental results. In Section II, we briefly introduce Rhapsody, model checking, CBMC and SPIN. We present our approach in Section III, the experimental results in Section IV and we conclude in Section V.

A. *Related work*

Schinz et al. [2] developed the first ever Rhapsody UML verification tool in 2004. It seems that Rhapsody allowed programmatic access during those days for this tool interfaced directly with Rhapsody; a feature not available in the current version(s) of Rhapsody. This tool supports a restricted set of UML constructs and needed manual assistance to successfully verify their test models. In particular, the authors manually constructed over-approximated models of the actual models to suit verification, fixed the configuration of some components, removed File I/O and so on. In contrast, our approach supports all features of UML, is fully automated and verifies the actual model as opposed to an over-approximation, thus avoiding the problem of having to refine the abstraction in case of a spurious counterexample. Further, the tool in [2] could verify models containing 9-to-24 states and our tool verified models containing 11-to-31 states.

There was another Rhapsody verification attempt [10], which works quite similarly to the tool in [2]. This tool translates Rhapsody models into the intermediate format of IF [3] and uses the backend analysis tools of the IF framework for state reachability verification. In this work too, the authors had to remodel their entire case study and manually prune many parts of their subject model in order to successfully verify. There have been quite a few other attempts at verification of UML models similar to Rhapsody, such as VERTAF [1] and LSC Verification [4]. We find that all these focus on a specific subset of features for the verification and need lots of manual assistance. In comparison, our approach is fully automated and scales better.

II. Background

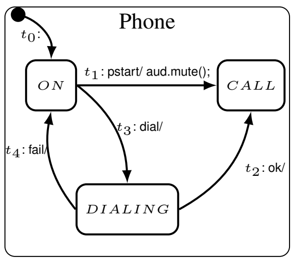
Here, we present a very high level description of Rhapsody statecharts and the model checkers CBMC and SPIN.

A. *Rhapsody statecharts*

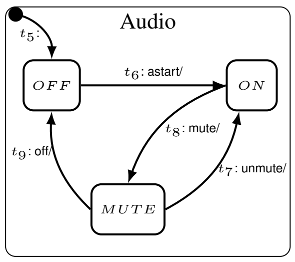
IBM's Rhapsody framework supports entire UML. However, the semantics of Rhapsody statecharts [5] are different from that of UML statecharts. Interested readers may refer to [17] to understand the differences. Here, we brief Rhapsody statecharts and their semantics. Rhapsody statecharts are extensions of conventional labeled state transition systems. The main extensions are hierarchy, concurrency and object-oriented communications and actions. The label of a transition is specified in the form **e[c]/a**, where **e** is an event (formally called *trigger*), **c** is a condition (formally called *guard*) and **a** is an action. For example in the statecharts of Fig 1, t_0 to t_9 are all transitions. Such a transition is enabled if its source state is active, event **e** occurs, condition **c** holds true and no higher priority transition is enabled. Transition exiting lower states in the hierarchy, get higher priority. Further, both **c** and **a** can be any valid C++ code. Rhapsody also allows C and Java code, but we restrict ourselves to the treatment of C++ code in this paper as all the models we deal with are in C++.

Fig 1 shows an example consisting of two Rhapsody statecharts, Phone and Audio, which respectively control a phone and an audio device. When the Phone and Audio statecharts start executing, they will start in the states ON and OFF respectively, as denoted by the source-less transitions t_0 and t_5. In Rhapsody, each event must be sent to some particular object. An event can't be broadcast. For example transition t_1 in Fig 1 sends the event *mute* to object *aud*, an Audio object. Upon receiving *mute*, if *aud* is not in ON state, it will simply drop that event as there is nothing to be done. On the other hand if *aud* is in ON state when *mute* is received, then it changes its state to MUTE by executing the transition t_8. All events raised by the system are stored in a event queue maintained internally by Rhapsody's execution environment. In each step, this environment reads the event at the head of the queue and sends it to its designated object. The environment then waits until the chain of reactions triggered by the event is completed. When the system reaches a state where in no more transitions can be executed, then the environment fetches the next event in the queue and dispatches it to its designated object.

The main execution semantics of Rhapsody consist of the notions of *event queue*, *step*, *microstep* and *null transition*.



(a) Statechart for Phone



(b) Statechart for Audio

Fig. 1. A Rhapsody Statechart Model

Rhapsody has an implementation of an event queue through which the statecharts communicate. Any event generated for any of the objects remains in this queue until Rhapsody's event dispatcher acts on it. Once the dispatcher dequeues the event and sends it to the designated object, then the object's statechart reacts to the event, marking the beginning of a *step*.

The execution of a step proceeds in a sequence of *microsteps*. The first microstep of a step is the one in which the reaction to the dispatched event is executed. If some more transitions get enabled as a result of this execution, then the second microstep takes place in order to execute the newly enabled transitions. If some more transitions get enabled due to the execution of the second microstep, then the third microstep takes place to execute those transitions and so on. Thus a sequence of microsteps is executed until there are no more enabled transitions. This marks the end of the step. During a step execution, the event dispatcher does not dispatch any event until all the microsteps constituting the step are completed. Once a step is completed, the event dispatcher dequeues and dispatches the next event in the event queue to the designated object, thus starting a new step. Therefore, only the first microstep of each step consists of a transition with a trigger(event) and rest of the microsteps consist only of transitions which do not have any triggers. Such transitions, which do not have triggers, are called *null transitions*.

For want of space, we can not describe the detailed semantics here. The reader may refer to [5] for the same.

B. Model checking

A model checker is a tool that checks a given model M for a desired property P across all paths in M. Here 'model' refers to any computation model including UML models, C code and even assembly code. There are many commercial and academic model checkers that check different kinds of models using a variety of techniques for various kinds of properties with varying degrees of success. For a comprehensive survey of model checking techniques and tools, the reader may refer to [14].

Of the freeware model checkers for C code, CBMC [12] and BLAST [18] are the most popular and robust choices. The SPIN [11] model checker also supports verification of C code embedded in its specifications. These tools have been around for more than a decade and are extensively used in the academia and the industry. We chose to experiment with CBMC and SPIN as they are better geared towards state reachability verification of the C code that we produce from Rhapsody models. CBMC and SPIN are freely available for download respectively at [8] and [9]. These also provide a comprehensive documentation about the respective model checkers, including theoretical background, practical applications and usage instructions.

III. Approach

Rhapsody statecharts are a visual formalism to model reactive behavior. While this formalism makes modeling easier, its verification is a difficult task. The reason, as we discussed earlier, is two-fold. The first problem is that Rhapsody allows C++ code as a part of modeling the behavior in the form of guards and actions. This mixture of code and state-transitions makes it difficult to create an automated verification framework for Rhapsody. The second problem is that Rhapsody does not provide automated access to its data, thus prohibiting a translation to any other formalism for which there are known verification tools or techniques. We describe below our approach to resolve these two problems and analyze reachability in Rhapsody models.

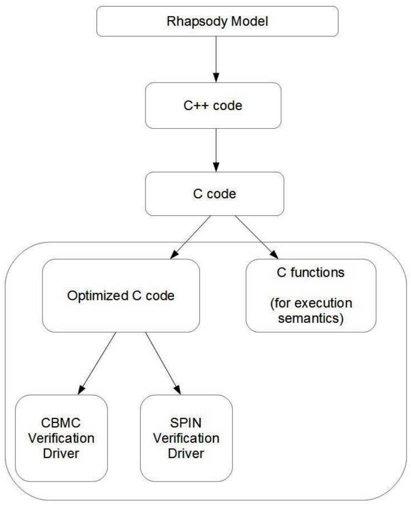


Fig. 2. Our approach description

Rhapsody framework has a code generator that generates C++ code for any given Rhapsody model. We first generate C++ code using the Rhapsody code generator and abstract (in some cases, eliminate) parts of the generated C++ code that do not affect reachability. We then translate this C++ code to C using the Edison Design Group (EDG [19]) C++ to C translator. In addition to this, we generate a set of C functions to encode the Rhapsody execution semantics succinctly.

Our encoding is similar in spirit to Microsoft's well known Static Driver Verifier framework [16], which verifies Windows device drivers for certain properties. This framework encodes the entire Windows kernel in C. For each Windows kernel API, the framework has a C function that faithfully represents the semantics of the Kernel API, but without the implementation details that are irrelevant for checking the properties of interest. This encoding is a part of the reason behind SDV's success [16]. In a similar spirit, we encode Rhapsody's execution framework with a faithful representation in C, but without the implementation details such as thread-spawning, mutex for thread-safety and so on. This encoding is a part of the reason behind the success of our attempt at the verification of Rhapsody statecharts. For instance, in order to implement the *step* semantics, we implement the event queue as a queue of (event, object) pairs.

The second source of our abstractions comes from being automotive domain specific. Typical automotive models consist only of global objects such as AudioObject, VideoObject and so on. The effect of their constructors can be precomputed and their destructors do not impact state reachability at all (as they get executed only during system termination). Therefore, we begun our optimizations by precomputing the effect of the constructors and then eliminating the constructors and destructors. We further removed those parts of the code that did not impact our reachability analysis. For example, there's a lot of animation code in the generated code files which is unnecessary if the code is not being used to animate the model in Rhapsody. The maximum benefit was achieved by replacing the code of Rhapsody's execution framework (called `OXF` in Rhapsody) with our own code that faithfully captures Rhapsody semantics. Our code is very concise as it does not get into the low-level details that the OXF needs to implement.

Finally, to verify the translated C code along with our functions that encode the execution semantics, we chose two model checkers - SPIN and CBMC. CBMC is a bounded model checker for C and therefore a natural choice. SPIN also seemed to suit the task as it allows embedding of C code in a Promela specification. Promela (**Pro**cess **Me**ta **La**nguage) [11] is a high-level language, supported by SPIN, to specify system descriptions.

The important aspects of our approach are illustrated below:

- **Queue Implementation** - Rhapsody implements the event queue by means of a set of ports where different objects send their event to. In the Rhapsody generated code, this *send* function is implemented differently for cases when the objects generates an event for itself and when it generates an event for some other object. This is so for each object which is a part of the model. This adds to the complexity of our analysis. Therefore, we wrote a much simpler implementation using arrays and defined standard queue operations (`enqueue`, `dequeue`, `isEmpty` etc.) on it.
- **Null Events** - Rhapsody disallows execution of null transitions if the number of times they execute in a row crosses a certain limit. This limit can be manually specified in the tool. *Null transitions*, as defined earlier, are transitions which do not have a trigger. The execution framework supports this by calling functions `pushNullTransition` and `popNullTransition`. These induce a chain of calls to functions in the `OXF` execution framework which are only needed for the internal book-keeping of `OXF`. We avoid all this by introducing the notion of a *null event* which is available only for a limited number (as specified) of null transitions executing in a row - as per the Rhapsody semantics.
- **Verification Drivers** for **SPIN** and **CBMC** - A microstep in Rhapsody executes by calling the function `rootState_processEvent` of each object in some sequence. These in turn call other auxiliary functions which are not important for our reachability analysis. For example, an object executes a microstep and sends notifications to other objects about some of the changes occuring in that microstep. These notifications are unimportant for our analysis as all these changes get captured in the state/data variables. The `OXF` framework does not allow the freedom of limiting the set of functions that execute in a microstep. We achieved this with our own implementation of (verification) drivers for SPIN and CBMC. Table 1 shows the pseudocode for our SPIN driver.

Algorithm 1 SPIN Verification Driver

```
c_code{ #include optimized_C_code };

/* variable declarations */

active proctype driver(){
 c_code{ initialize to default };
 if (queue is empty)
  create an event non-deterministically
  pick an object non-deterministically
  enqueue (object, event) pair
 else { skip; };
}

dequeue an event
dispatch event to the intended object
execute a step

set flag if desired state reaches
```

The algorithm above outlines the SPIN verification driver generated by our tool. This driver is generated in Promela. The `c_code{}` construct of Promela is what allows embedding C code in a Promela specification. Since *flag* is set when the desired state reaches, we attempt to validate the property that the *flag* variable never gets set. This can be encoded as an LTL (Linear Temporal Logic [20]) formula and SPIN allows encoding of such formulae in a promela specification. The CBMC driver, though generated in C, is structurally similar to the SPIN driver.

These optimizations failed to scale beyond some simple examples which we had manually created to test the approach. The industry models that we got from our clients were considerably bigger than the example models (see Section IV), causing an exponentially larger search space. We could reduce this by doing static analysis of the code using our in-house static analysis tool for C. The essential idea is to allow generation of only those events which can be immediately reacted upon, depending on the current state of each statechart. With this modification, although CBMC still ran out of memory, SPIN could verify reachability to all the states in one of the components of our model. Both the techniques, however, failed for a larger component.

In order to overcome this, we changed our approach and started looking at simulation to justify (un)reachability of states. The idea behind this was to restrict the behavioral branching at certain points during the execution, in order to force the system towards a desired state. As we were only looking to analyze reachability, it made sense to statically find the sequence of events leading to a desired state. However, one should note that such an analysis may not be feasible for a different class of properties such as repeated reachability.

For this analysis, we implemented a simulator which can feed events in the system to force certain behaviors (or, equivalently, paths). We compiled this guided simulator along with the Rhapsody generated C++ code to get the final executable. This approach scaled for all the components of our client model. The experimental results are described in the following section.

IV. Experimentation

We first experimented the model checking approach. The input to our tool is Rhapsody generated C++ code. The output of our tool is the optimized C code, a set of C functions to implement the Rhapsody `OXF` (execution) framework and the verification drivers for CBMC and SPIN. To test our approach, we first constructed eight simple models using various features of Rhapsody and C++. Then we tried our tool on these simple models, which both the CBMC and SPIN drivers generated by our tool verified successfully. Next, we tested our approach on five models from the automotive industry that correspond to five different components of an infotainment system(see Fig. 3): an FM Radio, a Next Generation Infotainment (NGI) System, a Camera, an Audio device and a Connection Manager which acts as a scheduler for resources shared among these components - a speaker, for example.

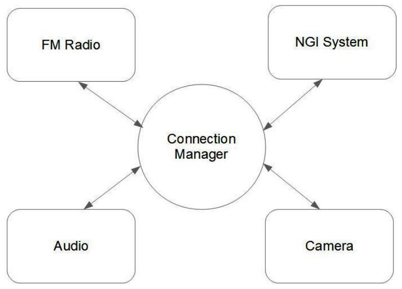


Fig. 3. Infotainment system

The CBMC and SPIN drivers, generated for these models by our tool, could not scale to the above industry models. To improve the scalability, we designed an optimization based on the following observation: in each *step*, any dispatched event is useless if it cannot be reacted to in the immediate *microstep* and, therefore, the verification drivers need not explore such events during their analysis. We enhanced our tool with this optimization by coding it in Java, using an in-house static analyzer for C code. This implementation first extracts the state-event relationship for the input model and uses this information to restrict the verification drivers in each *step* of the execution such that they explore only those events that the system can react to in the immediate *microstep*. With this optimization, we again ran our tool on the models. These experiments were performed on a 2.43 GHz Intel i5 processor with 2 GB primary storage (RAM). The analysis scaled to only one of the models (FM Radio, with SPIN driver), but failed on the rest. The experimental results are presented in Table I. The

Models	#states	LOC (Optimized C)	SPIN	CBMC
Simple Models	6-9	400-700	< 1 min	< 3 min
FM Radio	14	7540	< 4 min	OutOfMem
Others	11-28	39371-119725	OutOfMem	OutOfMem

TABLE I. Model checking experimentation data

analysis terminated in less than a few minutes on each instance for which it scaled. The models for which the analysis did not scale, both CBMC and SPIN drivers ran out of memory within a couple of hours. The CBMC driver exhausted the memory while trying to unwind for a depth of 1. (CBMC first unwinds the loops in the input program for a user-provided unwinding depth before checking for a property.)

We then implemented the simulation approach discussed earlier. This involved guiding the execution of the code for the models by analyzing the code further. In particular, we tried to extract the sequences of events leading to a particular state. This was quite helpful as we wanted to analyze only reachability and we could guide the code towards a desired state by providing only the necessary sequence of events. For this, we developed our own simulator generator in C++. This first analyzes the system to find the sequence of events that may lead to a desired state and then generates a simulator, in C++, that feeds the events to the system in the desired sequence. For each of our test models, we compiled the corresponding Rhapsody generated C++ code with the generated simulator C++ code to get the *final executable*. We experimented this approach on the simple models and the five industrial models. The experiments were performed on a 2.43 GHz Intel i5 processor with 2 GB primary storage (RAM). The guided simulation scaled for both the sets of models.

Models	#states	C++ LOC	Sim Gen	Sim Run
Simple Models	6-9	1632-2912	< 1 min	< 1 min
FM Radio	14	5874	< 2 sec	< 3 sec
NGI System	17	18683	< 3 sec	< 3 sec
Connection Manager	31	23561	< 3 sec	< 7 sec
Audio	11	24927	< 2 sec	< 5 sec
Camera	28	68135	< 4 sec	< 8 sec

TABLE II. Simulation Experimentation Data

Table II summarizes the results of this experiment. The column **C++ LOC** denotes the number of lines of C++ code (Rhapsody generated code and simulator). The columns *Sim Gen* and *Sim Run* respectively refer to the time taken for the generation of the corresponding simulator C++ code and the running time of the *final executable*. As can be seen from the table, the simulation approach demonstrated the reachability to all states in each model within a few seconds per model. The only exception was a state in the Connection Manager. Manual analysis revealed that the state was indeed unreachable. The reachability to this particular state was dependent on an event which was getting generated a step too early. As per Rhapsody semantics, events live exactly for one *microstep* once they are dispatched to their destination object and hence cannot be sensed in the next microstep. We reported this unreachability to the developers as this could have been a source of error. We later came to know that it was indeed erroneous and the Connection Manager was corrected to rectify the error.

V. Conclusion

We developed and implemented two methods towards automated verification of Rhapsody statecharts for state reachability. Our model checking based approach did not scale well to the industry models we experimented with, due to the complexity of the models. The simulation based approach scaled as it avoids having to deal with the complexity of the models and relies on a guided event generation strategy.

In our model checking approach, SPIN scaled better than CBMC. SPIN and CBMC implement different model checking techniques and are also engineered differently. We are not sure what would have happened if we had experimented them on a different set of models. It is plausible that, with additional abstractions, both SPIN and CBMC would have scaled. Coming up with such abstractions is an interesting future work.

The power of model checking lies in its ability to check all the computation paths in the system to be verified, which is also its bane for scalability. The power of simulation lies in its ability to focus only on generation of sensible inputs and thus avoiding the complexity of the system to be analyzed. However, it is often hard to find input sequences to guide the simulation. It seems like one needs a fine combination of these two approaches. For instance, once may first run simulation to prove some of the desired properties, then use the simulation results to prune the model and then employ model checking techniques to verify the pruned model for the rest of the properties. We are currently working on such an approach in the context of statecharts. Extending our work for the verification of properties other than state reachability is also an interesting line of research.

References

[1] P. Hsiung, S. Lin, C. Tseng, T. Lee, J. Fu, W. See. VERTAF: an application framework for the design and verification of embedded real-time software. In IEEE Transactions on Software Engineering, vol. 30, Oct. 2004, pp. 656-674

[2] I. Schinz, T. Toben, C. Mrugalla, and B. Westphal. The Rhapsody UML Verification Environment. In Proceedings of the Software Engineering and Formal Methods, SEFM 2004, pp. 174-183

[3] M. Bozga, S. Graf, I. Ober, I. Ober and J. Sifakis. The IF Toolset. In Proceedings of the Software Engineering and Formal Methods, SEFM 2004, pp. 237-267

[4] B. Westphal. LSC Verification for UML Models with Unbounded Creation and Destruction. Electronic Notes in Theoretical Computer Science. 144, 3, Feb. 2006, pp. 133-145

[5] D. Harel and H. Kugler The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML In Integration of Software Specification Techniques for Application in Engineering, 2004, pp. 325-354

[6] E. Gery, D. Harel, and E. Palachi. Rhapsody: A Complete Life-Cycle Model-Based Development System. In Proceedings of the Third International Conference on Integrated Formal Methods, 2002, pp. 1-10.

[7] IBM Rational Rhapsody. URL http://www.ibm.com/developerworks/rational/products/rhapsody/

[8] CBMC download site: http://www.cprover.org/cbmc/.

[9] SPIN download site: http://spinroot.com/spin/whatispin.html.

[10] OFFIS. Correct Development of Real-Time Embedded Systems: Project case studies. URL http://www-omega.imag.fr/cs/IAI/IAI.php

[11] G. Holzmann. Spin Model Checker, the: Primer and Reference Manual (First ed.). 2003. Addison-Wesley Professional.

[12] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004.

[13] A. Kulkarni., R. Metta, U. Shrotri, R, Venkatesh. Scaling up Model-Checking, A Case Study. In Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems. Proceedings of the GM R&D Workshop, Bangalore, India. (2007)

[14] Jhala, R., and Majumdar, R. Software model checking. ACM Computing Surveys., 41(4). 2009. pp. 154

[15] Y. Choi. From NuSMV to SPIN: Experiences with model checking flight guidance systems. Formal Methods in System Design. 30, 3. June 2007. pp. 199-216.

[16] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: static driver verification with under 4% false alarms. In Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (FMCAD '10). pp. 35-42.

[17] M. L. Crane and J. Dingel. UML vs. classical vs. rhapsody statecharts: not all models are created equal. In Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems, MoDELS 2005, p. 97-112.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST In Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN), LNCS 2648, Springer-Verlag, pages 235-239, 2003.

[19] **E**dison **D**esign **G**roup. http://www.edg.com/index.php?location=c_frontend

[20] A. Pnueli. The temporal logic of programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science. 1977. IEEE Computer Society Press, 4657.