SPIN or LURCH:
A Comparative Assessment of Model Checking and Stochastic Search for Temporal Properties in Procedural Code

John D. Powell
Jet Propulsion Laboratory
Pasadena CA USA
John.Powell@jpl.nasa.gov

David Owens
NASA IV&V Facility
Fairmont WV USA
owen@freeshell.org

Tim Menzies
Portland State University
Portland OR USA
tim@ menzies.us

Abstract

The difficulty of how to test large systems, such as the one on board a NASA robotic remote explorer (RRE) vehicle, is fundamentally a search issue: the global state space representing all possible behaviors of a complex software system is exponential in size. This state space explosion problem has yet to be solved, even after many decades of work. Randomized algorithms have been known to outperform their deterministic counterparts for search problems representing a wide range of applications. In the case study presented here, the LURCH randomized algorithm proved to be adequate to the task of testing a NASA RRE vehicle. LURCH found all the errors found by an earlier analysis of a more complete method (SPIN). Our empirical results are that LURCH can scale to much larger models than standard model checkers like SMV and SPIN. Further, the LURCH analysis was simpler than the SPIN analysis. The simplicity and scalability of LURCH are two compelling reasons for experimenting further with this tool.

1. Introduction

As software grows increasingly complex, testing becomes more and more challenging. Automatic testing by model checking has been effective in many domains including computer hardware design, networking, security and telecommunications protocols, automated control systems and others [2, 4, 6]. In this case study we will examine a model of a resource arbitration (RA) system aboard a NASA robotic remote exploration (RRE) vehicle. The model as built by automatic translators from design specifications is too large for the available model checking tools. The difficulty of how to test large systems, such as the one on board the RRE, is fundamentally a search issue: the global state space representing all possible behaviors of a complex software system is exponential in size. This state space explosion problem has yet to be solved, even after many decades of work [4].

LURCH, an approximate (not complete) alternative to traditional model checking based on a randomized search algorithm is being applied to the RA system on board the RRE in this case study

The study will make a determination about LURCH's potential benefits and limitations while testing a complex real world system. Randomized algorithms like LURCH have been known to outperform their deterministic counterparts for search problems representing a wide range of applications [7].

The cost of randomized algorithms is their inaccuracies. If complete algorithms terminate, they find all the features, and flaws therein, for which they are searching. On the other hand, by their very nature, randomized algorithms can miss important features / flaws. Past LURCH experiments suggests that this inaccuracy problem is not too serious. [13]

In the case study presented here, LURCH's random search found the same errors that were detected by SPIN. At the time of their discovery, using SPIN, the core causes of many of the errors in the RRE model were miss-diagnosed. The correct core cause would later become readily apparent while using LURCH. Also, this case study strongly suggests that LURCH can scale to much larger models than standard model checkers like SMV and SPIN. Thus, results of previous LURCH experiments [13] were confirmed by this study.

While we prefer the complete search of SMV and SPIN, some models are too large to be processed by these standard methods. If the choice is random search versus nothing at all (because the model is too big), the results of this case study suggest that random search methods like LURCH can still be a useful analysis tool.

2. The RA-RRE Model

The model used for the LURCH testing case study described in this paper is a model of a RA system on board a RRE vehicle and is referred to as the RA-RRE model throughout this paper.

The RA-RRE model used in this case study is a product of a research and technology development (R&TD) effort that was undertaken at the Jet Propulsion Laboratory (JPL). The RA-RRE model is specified in Stateflow® and consists of two identical User (User_1 and User_2) state charts (processes) that make requests for RRE resources used during operation through a message queue. The User processes run concurrently with an arbiter process (state chart), which processes the requests, taken from the message queue, made by the users. (*See Figure 1*) The arbiter will Grant, Deny, Pend, Rescind or Deny and Rescind a user request or recognize a message from a user as nonsense and ignore it. The appropriate arbiter response is sent back to the user making the request.

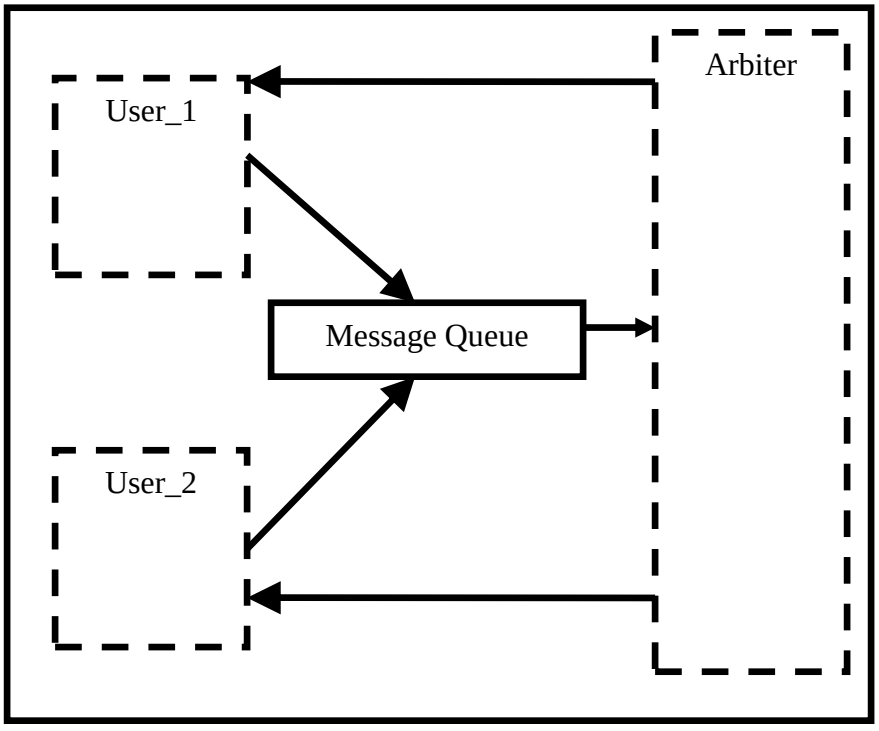

Figure 1: Concurrent RA-RRE Model State Charts / Processes

2.1. Relevant Stateflow Semantic

The Stateflow semantic that is most relevant to the testing results to be discussed is the fact that an ordering is imposed on the execution of

concurrent states specified in Stateflow state charts. Concurrent states are represented as "dashed boxes" in Stateflow. (*See Figure 1*) Concurrent states in a system are regarded as interleaving processes. This means that the steps in a given concurrent process may be executed in between the steps of any other concurrent process and vise versa. However, in Stateflow, execution of concurrent processes in a state chart has a specific turn taking requirement imposed upon them that is preserved and repeated. The order in which the concurrent processes are allowed to progress is determined by which concurrent state (dashed box) appears higher in the diagram graphically. Thus in the RA-RRE model the order would proceed as follows: Arbiter, User_1, User_2, Arbiter, User_1, User_2… indefinitely or until termination / deadlock. This ordering was specifically noted by a numbering scheme in the Stateflow graphic specification and reflected in the behavior of the automatically generated model (*See Section 2.2*)

2.2. Automatic RA-RRE Model Generation

The R&TD effort at JPL attempted to automatically generate Promela models for use by the SPIN model checker from Stateflow state charts. A state chart to Promela translator called HiVy was specifically designed and developed to use Stateflow's internal representation of state charts as input and produce semantically equivalent Promela code as output.

The Stateflow specifications consist of (*See Figure 2*):

- Hierarchical state charts that indicate legal control flow based on the behavior and constraints of the system being model
- C code embedded within various states to facilitate complex internal system behaviors that affect future behavior in the control flow as execution progresses through the various states of the hierarchical charts.

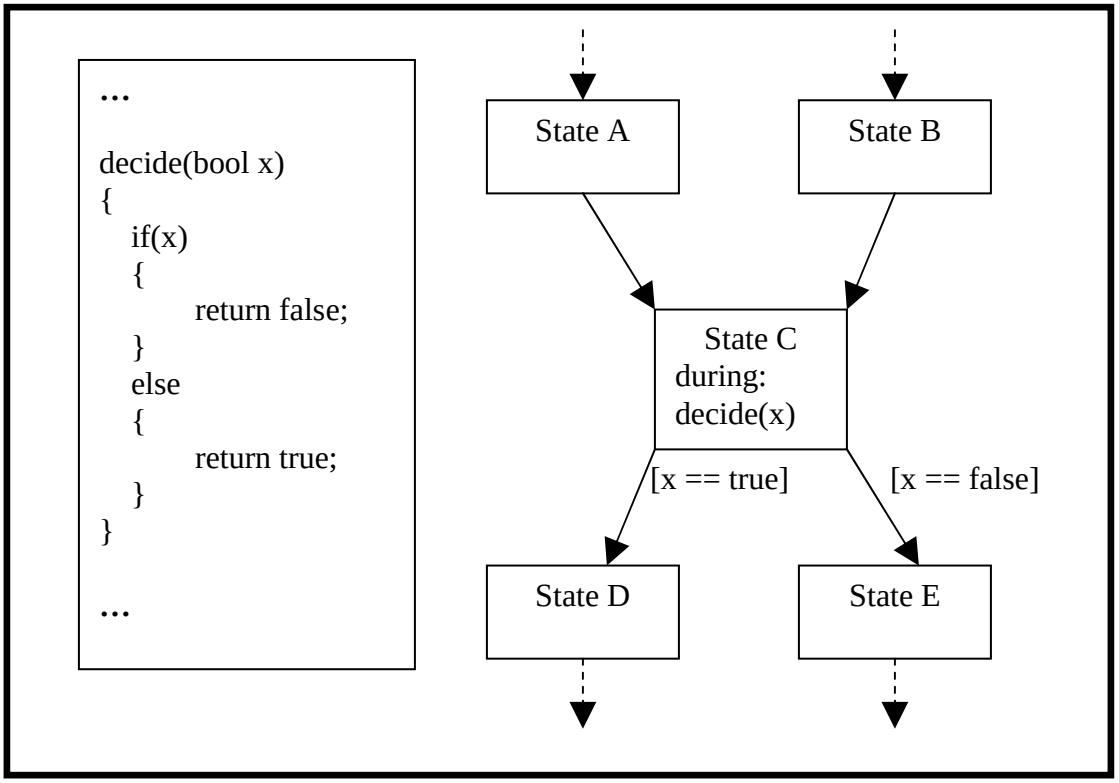

Figure 2: C code Embedded in a Stateflow State Cart

The HiVy translator output consists of Promela code that includes a concurrent process for entry into and execution at each level of hierarchically nested state charts. For example, a given hierarchical state chart in Stateflow was nested three levels deep then

- Three process would be created (one for each level) and
- An additional process would be created for each state chart residing at a given level and
- Additional processes created recursively for each process at the next lower level until every state chart at every level has its own concurrent process.

A complex semaphore locking system is included with a number of control processes that dictate the behavioral relationships between concurrent state charts that reside at a single level anywhere in the hierarchy.

While the HiVy automatic translation maintains fidelity to the behavior expressed in the Stateflow specification it introduces unacceptable overhead in terms of state space explosion. This results in models that are too large to be verified by SPIN within reasonable memory usage constraints.

The LURCH version of the RA-RRE model took on a somewhat different form due to the particulars of the LURCH modeling language. The LURCH modeling language allows a much closer coupling (than did Promela) between the model elements and the underlying C code that is embedded in the Stateflow state charts. The LURCH model essentially became a means of annotating the C code at a higher level to control legal calling sequences of the system model's actual C code while prohibiting illegal calling sequences as defined by the model.

2.3. RA-RRE Optimization

The selection of the RA-RRE model was driven by the factor that the state space of the model after optimization was still too large to be verified by the SPIN model checker within reasonable memory constraints. An optimization scheme for HiVy generated Promela models was developed and subsequently automated to reduce the size of the models' state spaces. The optimization algorithm targets the inordinate number of concurrent processes generated by the automatic translator and consolidates them. This resulted in a 44% increase in the efficiency of the models with regard to memory usage. While this allowed some large models to be subsequently verified other remained too large. Further investigation indicated that models with more C code embedded in it responded less dramatically to the optimization. This is due to the fact that the optimization scheme only addresses inefficiencies at the Promela level. SPIN was unable to verify models, such as the RA-RRE model, with large amounts of C code embedded below the Promela and/or state chart model level.

3. Testing of the RA-RRE Model

The SPIN verification and subsequent LURCH testing of the RA-RRE model revealed a conflicting requirement that could not be resolved within the timeframe and resources of this case study. However, it has provided fruitful ground for follow-on work with current and future NASA software projects that utilize Stateflow as a model based software development tool in the future. The requirements conflict that manifests during in the Stateflow specification of the system is precipitated by specific Stateflow state chart semantics and thus represents a class of design issues that must be addressed any time Stateflow is used to develop software systems in a model based development fashion. (i.e. automatic generation of software code via Stateflow state charts using the Matlab / Simulink suite.

3.1. SPIN Verification of the RA-RRE Models

The verification of the RA-RRE model with SPIN was performed over six variations of the model. First, after illogical results were obtained from initial SPIN simulation runs, it was determined that a hand edited addition of a single line at a specific point in both Promela models alleviated a deadlock condition without affecting the semantic meaning of the model. The deadlock condition that existed is believed to have been the result of a Promela syntax issue within the HiVy generated Promela model. Thus, two versions of the model were created: 1) RA-RRE (the original translation) and 2) RA-RRE-V2 (the hand corrected translation). Next, there are two versions HiVy translator (HiVy and HiVy_Eff) The HiVy version is the original Stateflow to Promela translator. The HiVy_Eff version is a modified version of the translator that is believed to produce a slightly more efficient Promela translation than the original HiVy version. By using both of the translators on the corrected RA-RRE-V2 model RA_RRE_Eff-V2 was added as a third model variation. Finally the automated optimization algorithm was applied to each existing variation of the model to maximize state space reduction. This yielded the six models that had to be verified:

- RA-RRE
- RA-RRE-Op (Optimization algorithm applied as a post processor)
- RA-RRE-V2 (Syntax correction to RA-RRE)
- RA-RRE-V2-Op (Optimization algorithm applied as a post processor
- RA-RRE_Eff-V2
- RA-RRE_Eff-V2-Op (Syntax correction to RA-RRE)

All six model variations would have to be put though verification to:

- Ensure that the hand correction did not affect the behavior of the model in a significant way
- Determine if the HiVy_Eff translator and/or the Optimization scheme implemented as a post process would impact the model state space enough to make SPIN verification possible.

The fact that similar results for basic deadlock and property verifications was the same for RA-RRE and RA-RRE-V2 suggests that the minor hand edit did not affect the semantics of the model behavior. (*See Table 1, Section 6*) However, SPIN reported two STATUS_ACCESS_VIOLATIONs in during each verification for both RA-RRE and RA-RRE-V2 before continuing to let the verification run indefinitely (each verification had to be manual aborted after 20 minutes). A STATUS_ACCESS_VIOLATION refers to the model attempting to access an unavailable or un-initialized memory address. After reporting these errors any verification result or the absence of one in this case, is unreliable and virtually meaningless.

The optimized versions of these two model variations (RA_RRE-Op and RA_RRE_V2-Op) yielded the same results as their un-optimized counterparts. Again because of the existence of runtime errors little if anything can be asserted based on these results.

The verification of the models variations generated by the efficient version of the HiVy translator (RA-RRE_Eff-V2 and RA-RRE_Eff-V2-Op) yielded different results from the four previously variations discussed above. Both found a deadlock at a very shallow search depth (depth 3). This indicates that the system need only take tree steps to find a sequence in which the systems processes all find themselves in a perpetual state of waiting on each other. Examination of the SPIN counter example showed that the processes either:

- Initialize and then a particular process takes a single step forward first in its execution behavior and all the other processes began waiting for notification while the active process waited on them for notification. As a result the entire system deadlocked.
- When the model is hand adjusted in a way that alleviates the above condition in order that the model could be explored at greater depths, the deadlock condition was found a

deeper depth. This counter example showed that the users (User_1 and User_2) would always overwhelm the arbiter eventually.

3.2. LURCH Testing of the RA-RRE Models

The LUCH testing results ultimately uncovered a set of related conflicts in the RA-RRE specification. First, LURCH discovered the same deadlock. Recall that the Stateflow semantics for concurrent state charts within a specification must execute in a prescribed turn taking order. In the case of the RA-RRE model this means that regardless of the graphical configuration of the Stateflow specification two user requests will always be generated for every one arbiter servicing of a request. The deadlock conditions seen in LURCH testing results over these models, when taken together as a class of related property violations, shows that the users overwhelm the "message queue" that is being used to store requests. When this occurs both users become deadlocked in states where they are waiting for disposition of pending requests for resources. In the RA-RRE model, this is indicative of and caused by an overflow of the messages queue that leaves the processes waiting:

- Each user is waiting in the pending state for the queue to become "un-full" so it can send the next message.
- Since the user processes are unable to reach the idle state, where it can listen for and received messages from the arbiter (*See Bullet Above*), they are unable to receive the arbiter's message thus the arbiter can not disposition any requests.

Therefore, the users are simultaneously and perpetually waiting for the Arbiter to return to is "Wait_for_Message" state, while the Arbiter is perpetually waiting for the users to return to their respective "Idle" states. This results in system deadlock

4. LURCH versus SPIN Modeling Complexity

According to Holzman (personal communication) embedding C code in to Promela for use by the SPIN model checker is very complex and has a steep learning curve, potentially taking several months of training to accomplish successfully. LURCH's means of embedding C code into the model is straight forward. Powell was able to successfully model a real world system specification that had considerable C code embedded in it after only 15 hours of informal LURCH training. (*See Table 1, Section 6*)

Verification of the embedded C code portions of the system model with SPIN is more problematic than with LURCH testing. When verifying a model that contains embedded C code with SPIN violations arising from incorrect C code are reported as a violation in a state at the model level only. Thus, it is difficult to quickly pinpoint where underlying C code errors are causing faults to occur. This black-box-like treatment of embedded C code caused the case study engineer to misdiagnose approximately 50% of the problems reported by SPIN as being caused by improper usage of SPIN constructs used for embedding C into Promela. LURCH testing would later reveal that in fact half the problems SPIN was reporting were caused by error with regard to unprotected accessing of fields from a C pointer when the pointer was null. (*See Table 1, Section 6*) After correcting many of the unprotected accesses half of the SPIN verification runs that reported problems during verification ran problem free until SPIN exhausted its memory resources. LURCH has shown promise in this study as an effective C code testing and debugging tool for:

- Models that contain complex embedded C code that must run correctly before valid and reliable results from exhaustive model checking verification can be obtained.
- Models that are too large for SPIN to successfully verify

While it is conceivable that the same C code operation information obtained from LURCH testing could be obtained from SPIN, obtaining the information from SPIN involves an indirect and more tedious process.

5. LURCH and SPIN Memory Utilization Performance

Our case is three-fold. First, in this case study, LURCH found the same bugs as SPIN. Second, LURCH seems a simpler method of adding temporal logic constraints to C code than SPIN. Given the complexities of the SPIN/C interface, we feel that this conclusion holds more generally than just this case study. Third, not only can LURCH be simpler than SPIN for finding the similar faults for C-based code, it also could scale to much larger systems than SPIN. This section offers the empirical evidence for this third conclusion.

To understand how SPIN and LURCH scale to larger models, four models were chosen where we could automatically vary the size of each model These four models were chosen since we could automatically generate larger variants of each one. The x-axis of *Figure 3* shows the size of each model and the y-axis shows (in a logarithmic scale) the time required by LURCH and SPIN to find the errors. The vertical dotted line in each plot of Figure 3 shows marks the limits to a direct comparison of LURCH and SPIN: at these boundaries, the models grew so large that the no SPIN mode could find the errors. The key observation to be made here in the zone where LURCH and

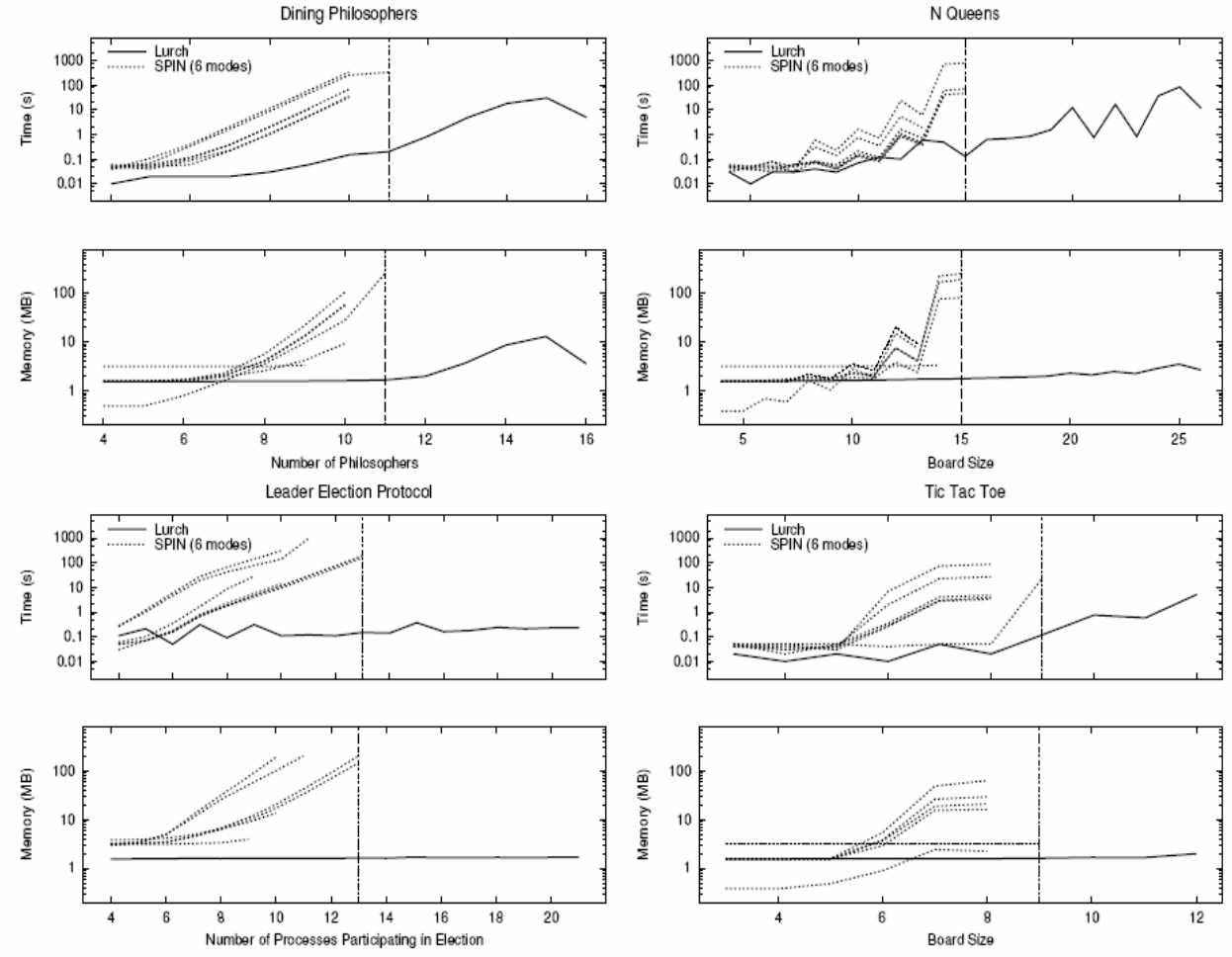

Figure 3: Figure 3: LURCH and SPIN Scalability Comparisons

SPIN can be compared (to the left of the dotted lines), LURCH found all the errors found by SPIN.

Figure 3 also comments on the relative scalability of LURCH's nondeterministic search to SPIN's more complete search. As model size grows, SPIN's search takes exponentially more resources. The resources required for LURCH are far more modest. In particular, LURCH's memory increases very gradually as problem size increases. This suggests that for very large problems, LURCH's nondeterministic search is not less safe than a deterministic search since LURCH can terminate on large problems that would defeat deterministic search.

With regard to the discovery of model errors SPIN and LURCH both found the same temporal violations (Deadlocks). SPIN is unable to full search the state space beyond the initial deadlock to find robust variations on various property violation due to state space explosion problems. LURCH's ability to scale up to larger model inputs, albeit with incomplete search methodologies, surpasses SPIN's. (*See Figure 3*) While both tools found the same logical error, LURCH's ability flexibility and ease of instrumentation made full diagnosis of the root cause of the error. Namely the conflicting requirements between the design assumption of the embedded C code and the Stateflow semantic regarding concurrent states that is discussed above. (*See Section 3.2*)

6. Conclusion

In this case study LURCH performed comparably with SPIN in that it found the same problems in the specification that SPIN did. Although SPIN's verification results, when possible to obtain, are close to full system verification, LURCH, due to its random, non-complete search strategy was able to test properties that overwhelmed SPIN more complete full verification strategy. The usability of LURCH to discover and diagnose problems in the system specification (Stateflow State Charts) showed great promise. With only 15 hours training, effective testing and diagnosis of problems within the RA-RRE system specification was performed. (*See Table 1*)

	SPIN	LURCH
Finding Errors – Dead Lock	Found Deadlock Condition	Found Deadlock Condition
Finding Errors – Property Verification	Model was too Large to Verify Properties	Found multiple variations on Deadlock over properties
Embedded C code	Steep learning curve	Easily Accomplished with minimal training
Diagnosis of Cause of Model Errors	Masked errors in embedded C code as syntactic / semantic problems embedding C into Promela	Easily instrumented to provide visibility into embedded C code errors. This lead to discovery of error relating to fundamental system specification conflicts

Table 1: Summary Comparison of SPIN and LURCH capabilities during the Case Study

These results encourage us to experiment further with LURCH.

7. Acknowledgement

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] P. Cheeseman, B. Kanesfy, and W. Taylor. Where the Really Hard Problems Are. In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence IJCAI{91, Sidney, Australia, 1991.
[2] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A New Symbolic Model Checker. International Journal on Software Tools for Technology Transfer, 2(4), 2000.
[3] E. Clarke, O. Grumberg, and D. Long. Verifiction Tools for Finite-State Concurrent Systems. A Decade of Concurrency|Reections and Perspectives, 803, 1993.
[4] E. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, Cambridge, MA, 1999.
[5] B. Hayes. On the Threshold. American Scientist, 91(1), 2003.
[6] G. Holzmann. The Model Checker SPIN. IEEE Transactions on Software Engineering, 23(5), 1997.
[7] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. In Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference, 1996.
[8] T. Menzies and B. Cukic. Adequacy of Limited Testing for Knowledge-Based Systems. International Journal on Artificial Intelligence Tools, 9(1), 2000.
[9] T. Menzies, D. Owen, and B. Cukic. Saturation Effects in Formal Verification. In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), 2002.
[10] R. Motwani and P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995, (reprinted 1997,2000).
[11] D. Owen. Random Search of AND-OR Graphs Representing Finite{State Models. Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University, 2002.
[12] D. Owen and T. Menzies. Lurch: a Lightweight Alternative to Model Checking. In SEKE '03, 2003.
[13] David Owen and Tim Menzies and Mats Heimdahl and Jimin Gao. On the Advantages of Approximate vs. Complete Verification: Bigger Models, Faster, Less Memory, Usually Accurate; IEEE NASA SEW 2003.
[14] Paula J. Pingree, Erich Mikk, Gerard J. Holzmann, Margaret H. Smith, Dennis Dams, validation of mission critical software design and implementation using model checking. IEEE DASC, Oct 2002
[15] J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. In Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering, number 1687 in LNCS, pages 163{179, September 1999.
[16] J. M. Thompson, M. W. Whalen, and M. P. Heimdahl. Requirements capture and evaluation in Nimbus: The light-control case study. Journal of Universal Computer Science, 6(7):731{757, July 2000.
[17] M. W. Whalen. A formal semantics for RSML_e. Master's Thesis, University of Minnesota, May 2000.