On the Advantages of Approximate vs. Complete Verification: Bigger Models, Faster, Less Memory, Usually Accurate

Mats Heimdahl, Jimin Gao
Dept. of Computer Science & Engineering
University of Minnesota
Minneapolis, MN 55455
heimdahl@cs.umn.edu
jgao@ece.umn.edu

David Owen,Tim Menzies
Lane Dept. of Computer Science
and Electrical Engineering
West Virginia University
Morgantown, WV 26506
drobo75@hotmail.com
tim@menzies.us

1 Introduction

As software grows increasingly complex, verification becomes more and more challenging. Automatic verification 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]. Many real-world software models, however, are too large for the available tools. The difficulty—how to verify large systems—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].

We have been exploring LURCH, an approximate (not necessarily complete) alternative to traditional model checking based on a randomized search algorithm. 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 are their inaccuracies. If complete algorithms terminate, they find all the features they are searching for. On the other hand, by their very nature, randomized algorithms can miss important features. Our experiments suggest that this inaccuracy problem is not too serious. In the case studies presented here, LURCH's random search usually found the correct results. Also, these case studies strongly suggest that LURCH can scale to much larger models than standard model checkers like SMV and SPIN.

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), our results suggest that random search methods like LURCH can still be a useful analysis tool.

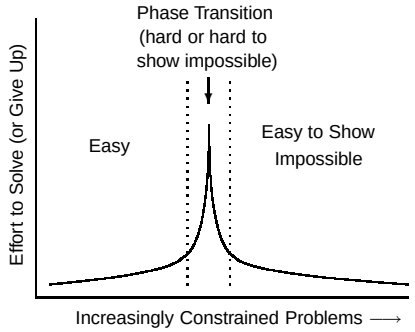

Figure 1. Hard problems exhibit a phase transition.

2 The Phase Transition

Difficult search problems, e.g., *NP-hard* problems, have been shown to exhibit a *phase transition* (figure 1) [1, 5, 8]. In some cases the problem turns out to be very easy to solve; other cases are impossible. For these impossible cases, however, it usual easy and fast to show that they can not be solved.

So there are easy cases and cases that can easily be shown to be unsolvable. Are there cases that are very hard but solvable? Or, for unsolvable cases, are there any that are very hard to determine that they are not solvable? Yes, these pathological cases exist,

but they are rare: there is just a narrow transition region where a lot of effort is required to either solve or determine that no solution is possible. This, in the words of Cheeseman et.al., is "where the *really* hard problems are" [1].

Figure 2 shows how a simple solution strategy can be used to exploit easy problems but avoid wasting effort on problems that are very hard or unsolvable [11]. We put a relatively small amount of effort into solving the problem with our simple strategy (effort could be time, memory, or some other limited resource). If the problem is easy, we solve it easily. If we do not solve the problem, we know it is either very difficult or impossible. Of course there is nothing revolutionary about this approach. The key point is that the phase transition region is narrow. A very simple strategy is therefore capable of solving very nearly everything that could be solved by much more sophisticated strategies, but with much less effort.

3 Approximate Verification

Model checking is used to verify that finite-state concurrent systems satisfy specified temporal logic properties [3, 6]. The amount of memory required to store all possible behaviors of a finite-state concurrent system is, in the worst case, an exponential function of the size of the original model—this is the so-called *state-space explosion* [4]. For many systems, model checking requires a prohibitively large amount of memory and time.

The full model checking technique may be overkill, however, for problems which turn out to be in the easily solvable range (recall figure 1). We have hence been experimenting with a simple, efficient alternative to model checking in a tool called LURCH. The algorithm is described in brief in figure 3 (for full details, see [9–11]. LURCH uses an memory-saving AND-OR graph representation of the composite system behavior[1]. LURCH's search space contains $A * V$ number of nodes: i.e. one node for each possible assignment A to a every variable V. By contrast, the search space of a model checker contains at most A^V nodes: i.e. one node each consistent set of assignments to all variables.

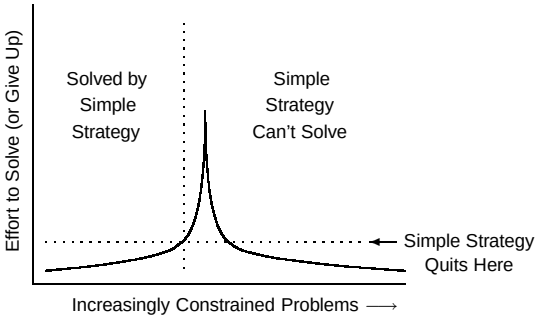

Figure 2. The power of a simple solution strategy.

```
1: next-global-state(state) {
2: Execute a transition for every machine in which there is
     at least one whose input conditions are satisfied; if more
     than one transition is possible for a machine, choose one
     at random. }

3: path(state) {
4: while (¬(path-end OR cycle)) do
5:    state ← next-global-state(state); }

6: main() {
7: repeat
8:    path(initial-global-state);
9: until (user-defined maximum reached) }
```

Figure 3. LURCH's partial, random search procedure.

LURCH's basic search procedure returns one global-state path traced through the composite system behavior, terminating whenever a dead end or cycle is found. In practice we have found that LURCH is able to explore a space more quickly if the cycle detection scheme is somewhat relaxed. In the current version, the LURCH continues even after the first repeat global state in a path (i.e., when a cycle is first detected); instead, the while loop is exited after n repeat global states, where n is a number input by the user. In this way LURCH is allowed to pursue intersections, i.e., places where a path may cross itself but then continue to find new information.

4 Flight Guidance System Experiment

To validate the performance and accuracy of LURCH in a realistic situation, we conducted an experiment using a model of the mode logic for a commercial flight guidance system (FGS) developed in collaboration between Rockwell Collins Inc. and the University of Minnesota. The mode logic is captured in RSML^{-e} and automatically translated to SMV and LURCH through NIMBUS, the development environment for RSML^{-e} [12–14].

[1] To justify the analogy between LURCH results and phase transition results reported by others, note that complete search of the AND-OR graph used by LURCH to represent the composite system is in fact NP-hard [10].

Models	Violations	SMV	LURCH				
			Test 1	Test 2	Test 3	Test 4	Test 5
FGS_Fault1	Property 1	442s/28.3MB	435s/19.0MB	191s/9.7MB	50s/3.9MB	93s/5.6MB	14s/2.7MB
	Property 2	442s/28.3MB	435s/19.0MB	191s/9.7MB	50s/3.9MB	93s/5.6MB	14s/2.7MB
FGS_Fault2	Property 3	214s/25.8MB	35s/3.4MB	47s/3.9MB	31s/3.3MB	19s/2.8MB	18s/2.6MB
	Property 4	214s/25.8MB	NF	NF	NF	NF	NF
FGS_Fault3	Property 5	1800s/40.3MB	NF	338s/16.1MB	NF	1711s/71.1MB	NF
	Property 6	1800s/40.3MB	110s/6.1MB	42s/3.7MB	47s/4.0MB	108s/6.0MB	68s/4.7MB
FGS_Fault4	Property 7	635s/28.1MB	0.4s/1.9MB	0.3s/1.9MB	1.2s/2.0MB	0.7s/2.0MB	0.4s/1.9MB
	Property 8	635s/28.1MB	131s/7.2MB	61s/4.4MB	86s/5.4MB	18s/2.8MB	36s/3.4MB

Table 1. Performance results for NuSMV and LURCH on the fault seeded FGS models.

4.1 Experimental Setup

The RSML^{-e} FGS model consists of 2564 lines of RSML^{-e} code defining 142 state variables. When translated to SMV, we get 2902 lines of SMV code that requires 849 BDD variables for encoding. The translated LURCH state machine model consists of 635 local states and 1288 transitions. The RSML^{-e} model has been extensively validated through testing and we have verified over 250 properties using NuSMV.

To validate the performance of the analysis tools, we had to create a collection of faulty specifications and select a subset of the 250 properties that would reveal the faults. To create the faulty specifications, we reviewed the revision history of the FGS model and identified a set of changes that had been made in response to faults found during our original verification effort—from this set we selected four faults for reintroduction into the model. We then hand seeded these actual faults back into the specification, thus creating four faulty versions of the FGS specification.

To select properties of interest, we reran the complete verification suite on the FGS using NuSMV. We found that 8 properties of the original 253 were violated in the four faulty specifications—these are the eight properties we have used in this experiment. By chance, two distinct properties were violated in each of the four faulty FGS models.

4.2 Results

LURCH was run with a search depth limit of 2000 and a "cut of" of 2000 seconds, that is, if no violation was found within 2000 seconds, the search was terminated. We could have used state space saturation (as discussed earlier in this paper) as a termination criteria. Given the enormous state space of the FGS, however, reaching saturation was not an option so we elected to simply select an ample, but tolerable, maximum search time. Finally, due to the random nature of LURCH, we ran each verification run five times on each FGS model for the set of properties. In both NuSMV and LURCH, all properties were grouped in a batch.

Table 1 summarizes the performance results of NuSMV and LURCH. Each entry shows the CPU time (in seconds) and peak memory usage (in megabytes) taken to find the particular property violation. The NuSMV results were obtained by running NuSMV with command options `-dynamic` (dynamic variable reordering) and `-coi` (cone of influence reduction). Without these options, NuSMV was unable to produce any result within 10 hours and without memory usage exceeding 900MB. We report the LURCH results for each of the five runs we performed for each specification. In the table, an NF entry indicates that LURCH did not find a counterexample for the property within our allotted search time of 2000 seconds. For the original FGS model, as expected, no property violations were found by either NuSMV nor LURCH and the results are not reported in the table.

Our results indicate that LURCH finds the same property violations as NuSMV in most cases. When LURCH finds a violation, it typically does so orders of magnitude faster than NuSMV and LURCH uses orders of magnitude less memory. This result is encouraging and indicates that LURCH could be a very powerful *refutation* tool to use when debugging large models—models that cannot be analyzed using current exhaustive verification techniques.

For some properties LURCH either failed to find a counterexample (Property 4) or only found it on some runs (Property 5). This inaccuracy is not surprising given the incomplete searches in LURCH. In fact, some members of the research team expected LURCH to be much less accurate than what we saw in this case study. Furthermore, by extending the search depth during "tinkering" with LURCH, the violation of Property 5 could also be consistently found in a reasonable amount of time[2]. Our experiences lead us to believe that the performance of LURCH can be improved by tuning its search parameters—in the experiment we used the de-

[2]The results gathered while informally experimenting with LURCH are not shown in Table 1 to keep the conditions unchanged for the formal experiment

fault settings. The effect of parameter tuning is largely unknown at this time; for example: will we get better performance through many shallow searches as opposed to a few long searches? Understanding and limiting the magnitude of the inaccuracy of LURCH on realistic models is a critical issue if the techniques implemented in LURCH are to be used as an approximation for full verification where the size of the model precludes full verification—this issue will be the subject of future investigations.

Finally, the search results for LURCH are surprisingly consistent—the variance in search time for a fault is rather small. In addition, LURCH seems to reliably find (or not find) violations of properties (recall that Property 5 could be reliably found with somewhat different search parameters). This lends some support for the hypothesis that if a fault is present it is either very easy *or* very difficult to find. What we hope to explore in future investigations is how many fault really fall in the category of "really difficult to find," regardless of how LURCH's adjustable parameters are set.

5 Conclusion

Preliminary results with a random search device for checking temporal properties have been very encouraging. We are hence motivated to test the generality of this work via future case studies.

Acknowledgements

We would like to thank Steve Miller, David Lempia, and Alan Tribble at Rockwell-Collins' Advanced Technology Center for building the FGS models and providing us feedback on our translation tools. This work was conducted at the University of Minnesota and West Virginia University. Partial support for this work came from the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility under NASA grant NAG-1-224 and NASA contracts NCC-01-001, NCC2-0979, NCC5-685. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

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, Austrailia*, 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. Verification Tools for Finite-State Concurrent Systems. *A Decade of Concurrency—Reflections 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] 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.

[11] D. Owen and T. Menzies. Lurch: a Lightweight Alternative to Model Checking. In *SEKE '03*, 2003.

[12] 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.

[13] 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.

[14] M. W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.