Finding Faults Quickly in Formal Models using Random Search

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

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

Abstract

As software grows more complex, automatic verification tools become increasingly important. Unfortunately many systems are large enough that complete verification requires a lot of time and memory, if it is possible at all. In our preliminary studies, random search, although not a complete technique, was able to find most faults significantly faster and with less memory than would be required for full verification. Here we present an experiment in which random search was used to find faults in fault-seeded models of a large commercial flight guidance system. To assess the performance of random search we compared it to a full verification done by the model checker NuSMV. The random search results were surprisingly complete, finding nearly 90% of the faults reported by NuSMV—and these results were generated faster and using less memory. We suggest that random search be used in conjunction with verification tools, perhaps as a fast debugging tool during model development, or even as an alternative model checking strategy on models for which the time and memory requirements would make full verification impossible.

1 Introduction

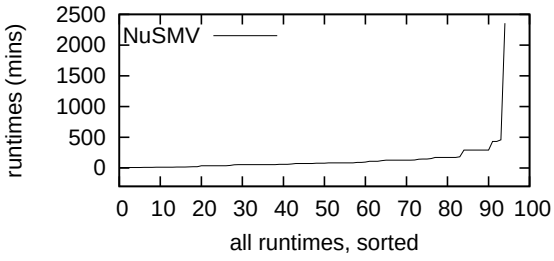
Automatic verification by model checking has been effective in many domains including computer hardware design, networking, security and telecommunications protocols, automated control systems and others [4, 8, 14]. Many real-world software models, however, are large enough that full verification requires much time and memory—if full verification is possible at all. Incomplete but faster state-space exploration techniques, capable of finding errors but not formally proving their absence, are useful for faster feedback during development. We have implemented one such incomplete technique, random search of a compact AND-OR graph representing the state space, in a tool called LURCH.

In this paper we use LURCH to assess the performance of a random search strategy on a commercial flight guidance system model. First, we wanted to make sure that the random search worked quickly and without a large amount of memory for a large, real-world software model, since in the past we have observed the time and memory savings of random search only in artificially generated (and often highly symmetric) models [21]. Also, we wanted to determine whether random search can accurately and consistently find a significant portion of the actual faults reported by full verification.

To this end, we applied random search (using LURCH) and full verification (using the model checker NuSMV) to a collection of faulty models of the full-scale mode logic of a flight guidance system (FGS).[1] Figure 1 shows the execution time of NuSMV for 45 FGS models (with seeded faults) using a verification suite of 60 properties. The check usually took an hour or two and, in the worst case, took over 36 hours. These execution times are far too slow to support analysis of a specification as it is being developed—a phase where the specification is changing rapidly and the analysis is routinely run many times per day.

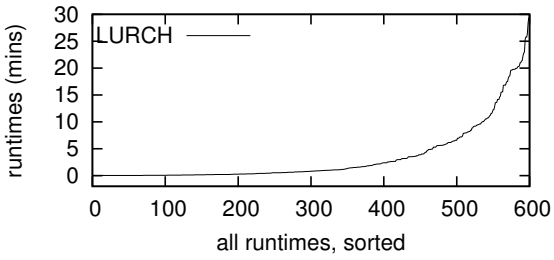
The time needed to find a property violation using random search (with LURCH) is shown in Fig-

[1] We thank Dr. Steve Miller and Dr. Alan Tribble of Rockwell Collins Inc. for the information on flight control systems and for letting us use the RSML^{-e} models they have developed using the NIMBUS tool from the University of Minnesota.



	median	mean	max
NuSMV	73	125	2358

Figure 1. Time (rounded to minutes) to find violations using NuSMV.



	median	mean	max
LURCH	1	3	29

Figure 2. Time (rounded to minutes) to find violations using LURCH.

ure 2 (the figure reports the 600 times a property violation was found—a detailed account of our experimental setup will be given later). In most cases property violations were found very quickly; few violations required the full 30 minutes we allocated for the random search. As to the accuracy and consistency of random search, LURCH found property violations accounting for nearly 90% of the faults found by full verification with NuSMV, finding 75% of the property violations reported by NuSMV in four out of five runs. Based on these observations, we believe techniques based on random search are effective debugging tools that should be applied to get quick results and identify faults with little effort.

The rest of this paper describes LURCH and the case study reported in Figure 1 and Figure 2. Section 2 describes LURCH, and section 3 describes the experiment done running LURCH on fault-seeded flight guidance system models. The final section discusses the external validity of our results.

```
1: next-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-state(state); }

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

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

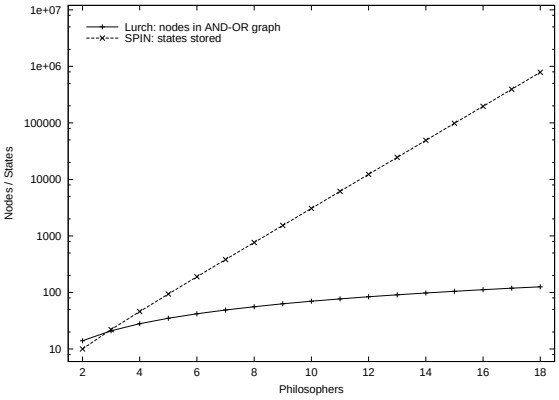


Figure 4. For Dining Philosophers problem models, size of LURCH's AND-OR graph vs. states stored by the model checker SPIN [14].

2 LURCH

2.1 Implementation

Model checking is used to verify that finite-state systems satisfy specified temporal logic properties [7, 14]. The amount of memory required to store all possible behaviors of a finite-state system is, in the worst case, an exponential function of the size of the original model. Thus for many systems model checking requires a large amount of memory and time. Our prototype random search tool, LURCH, uses a memory-saving AND-OR graph representation of the composite system behavior and the partial, random search algorithm summarized in figure 3 [18, 20]. Data shown in figure 4, from a preliminary study, shows the efficiency of the AND-OR graph for a range of models representing the Dining Philosophers problem; for these models

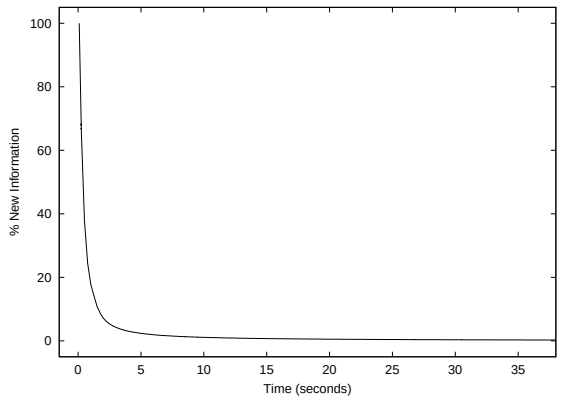


Figure 5. LURCH output for a typical model: quick saturation.

the number of states required for full verification by the model checker SPIN [14] increases exponentially with the number of philosophers, whereas the number of nodes in Lurch's AND-OR graph increases linearly.

Our random search algorithm is *partial* because, unlike the full model checking technique, there is no guarantee that all possible behavior is explored; the algorithm is *random* because the choice of which behavior to explore (or ignore) is nondeterministic. Our basic implementation is a *Monte Carlo* algorithm: the search procedure runs again and again, each time increasing the probability of finding a property violation.

In many cases we quickly find a violation, but for those in which none is found, how do we know when to stop? Figure 5 shows output from LURCH running on a typical input model. As LURCH runs, it explores the reachable state space, at first finding nearly all new information, but after a little while most of LURCH's findings are redundant; Figure 5 illustrates this: the percentage of the model explored which is new (vs. redundant) starts out at 100%, but quickly decreases to near zero. We use this quick *saturation* effect in LURCH output (see [18]) to determine when to stop: when some set saturation point is reached, we assume that LURCH is unlikely to find any more interesting information.

For very large input models, such as the flight guidance system described below, it is not always practical to wait for a low saturation percentage, so LURCH is terminated at a reasonable time or memory cutoff. We are continuing to experiment with different stopping criteria, so that LURCH can run as quickly as possible but with consistent results.

2.2 Additional Features of LURCH

To efficiently track which states have been reached LURCH stores hash values based on the names of all local states present in the state to be stored. Each state gets one integer; these are all kept in a tree, which remains approximately balanced because the hash values are evenly distributed across the range of integers. So in practice LURCH treats these hash collisions as repeat states although they are actually *potential* repeat states. LURCH allows the user to limit the amount of memory available for state storage.

LURCH's basic search procedure returns one 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, LURCH continues even after the first repeat state in a path (i.e., when a cycle is first detected); instead, the while loop in figure 3 is exited after n repeat 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.

LURCH simulates *synchronous* execution of finite-state machines in the input model; that is, at each step forward in time, every individual finite-state machine that is able to execute a transition does, and the order of these intra-time-step executions is considered irrelevant. Also, any side effects of a transition that would interfere with the state of things at the start of the time-step do not take effect until after all the machines have had a chance to go forward.

By adding a simple modification we can simulate *asynchronous* execution of the finite-state machines in the input model. Instead of allowing an arbitrary number of transitions to be processed at each time step, which would correspond to giving all machines a chance to move forward, we allow only one machine to transition forward at each time step. Side effects of that transition take effect before any other machines have a chance to transition forward, and the particular interleaving of machines' transitions is considered significant, as in an asynchronous system. For example, in the preliminary study associated with figure 4 we used LURCH in asynchronous mode to find deadlocks in models representing the Dining Philosophers problem.

3 Flight Guidance System Experiment

To validate the performance and accuracy of LURCH in a realistic situation, we conducted a large experiment based on a model of the mode logic for a commercial flight guidance system developed in collaboration between Rockwell Collins Inc. and the University of Minnesota. The mode logic is captured in RSML^{-e} [25], a fully formal synchronous specification language, and automatically translated to NuSMV and LURCH through NIMBUS [23], the development environment for RSML^{-e}. The results of that study were presented in brief in our introduction. Further details are offered below.

3.1 Background

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state, generating pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS can be broken into two parts: the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time; and the flight control laws, which accept information about the aircrafts current and desired state and compute the pitch and roll guidance commands. In this case study we check only the mode logic.

Figure 6 illustrates a graphical view of the FGS in the NIMBUS environment. The primary modes of interest in the FGS are the horizontal and vertical modes. The horizontal modes control the behavior of the aircraft about the longitudinal, or *roll*, axis, while the vertical modes control the behavior of the aircraft about the vertical, or *pitch*, axis. In addition, there are a number of auxiliary modes, such as half-bank mode, that control other aspects of the aircraft's behavior.

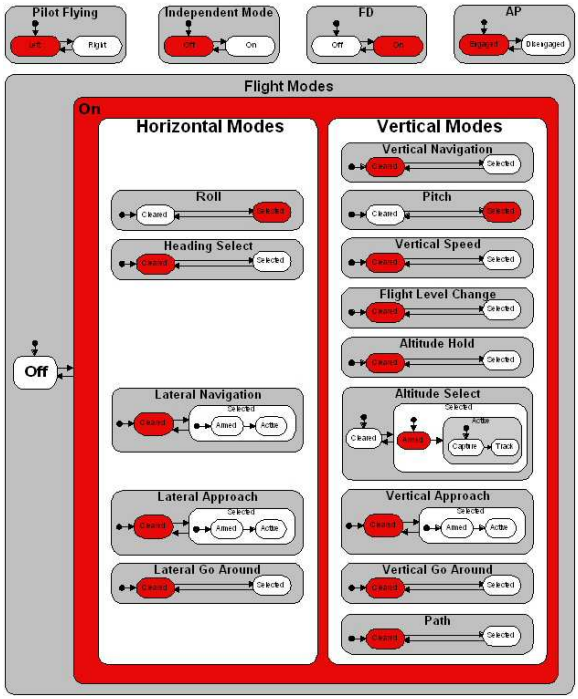


Figure 6. Flight Guidance System

3.2 NIMBUS and RSML^{-e}

Figure 7 shows an overview of the NIMBUS tools framework. The user builds a behavioral model of the system in RSML^{-e} (see below). After evaluating the functionality and behavioral correctness of the specification using the NIMBUS simulator, users can translate the specifications to PVS (a theorem-proving system), NuSMV, or LURCH input languages.

RSML^{-e} is based on the Statecharts-like [11] Requirements State Machine Language (RSML) [17]. RSML^{-e} is a fully formal and synchronous data-flow

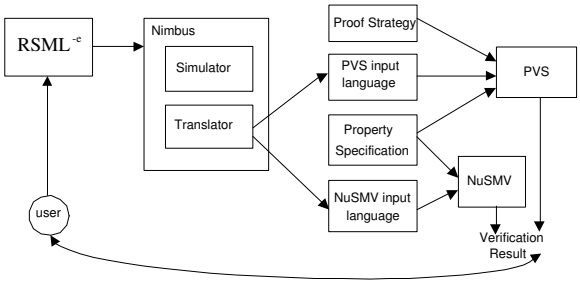


Figure 7. Verification Framework.

```
STATE_VARIABLE ROLL : Base_State
 PARENT          : Modes.On
 INITIAL_VALUE   : UNDEFINED
 CLASSIFICATION  : State
 TRANSITION UNDEFINED TO Cleared IF NOT Select_ROLL()
 TRANSITION UNDEFINED TO Selected IF Select_ROLL()
 TRANSITION Cleared TO Selected IF Select_ROLL()
 TRANSITION Selected TO Cleared IF Deselect_ROLL()
END STATE_VARIABLE

MACRO Select_ROLL() :
 TABLE
  Is_No_Nonbasic_Lateral_Mode_Active()    : T;
  Modes = On                              : T;
 END TABLE
END MACRO

MACRO Deselect_ROLL() :
 TABLE
  When_Nonbasic_Lateral_Mode_Activated()  : T *;
  When(Modes = Off)                       : * T;
 END TABLE
END MACRO
```

Figure 8. A small portion of the FGS specification in RSML^{-e} .

language without any internal broadcast events (the absence of events is indicated by the $^{-e}$).

An RSML^{-e} specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants; *input variables* are used to record the values observed in the environment, *state variables* are organized in a hierarchical fashion and are used to model various states of the control model, *interfaces* act as communication gateways to the external environment, and *functions and macros* encapsulate computations, which provides increased readability and ease of use.

Figure 8 shows a small portion of an RSML^{-e} specification of the Flight Guidance System.[2] The figure shows the definition of a state variable, `ROLL`. `ROLL` is the default lateral mode in the FGS mode logic.

The conditions under which the state variable's value changes are defined in the `TRANSITION` clauses of the definition. The condition tables are encoded in the macros `Select_ROLL` and `Deselect_ROLL`. The tables are adopted from the original RSML notation—each column of truth values represents a conjunction of the propositions in the leftmost column ('*' represents a "don't care" condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in disjunctive normal form.

[2]We use here the ASCII version of RSML^{-e} since it is much more compact than the more readable typeset version.

3.3 Experimental Setup

For our experiments we used the largest FGS model available to us—a model close to the production systems at Rockwell Collins Inc. The RSML^{-e} FGS model consists of 2,564 lines of RSML^{-e} code defining 142 state variables. When translated to NuSMV, we get 2,902 lines of code, requiring 849 BDD variables for encoding. The translated LURCH state machine model consists of 217 3-state machines, producing an AND-OR graph with 1,923 nodes. This RSML^{-e} model has been extensively validated through testing and we have previously verified close to 300 required properties using NuSMV.

To assess the performance and fault finding capability of LURCH, we created a collection of faulty specifications and selected a subset of the 300 properties that would reveal the faults. To create the faulty specifications, we first reviewed the revision history of the FGS model to understand what types of faults were removed during the original verification process. We then implemented a random fault seeder to inject representative faults to create a suite of faulty specifications. The faults we seeded fell into the following four categories:

Variable Replacement: A variable reference was replaced with a reference to another variable of the same type.

Condition Insertion: A condition that was previously considered a "don't care" (*) in one of the tables was changed to T (the condition is required to be true).

Condition Removal: A condition that was previously required to be true (T) or false (F) in a table was changed to "don't care" (*).

Condition Negation: A condition that was previously required to be true (T) in a table was changed to false (F), or vice versa.

We used our fault seeder to generate 100 faulty specifications (25 for each fault class). As an example, Figure 9 shows a missing condition fault contained in macro `When_LGA_Activated`, the fault was created by changing the table from requiring the Boolean variable `Is_This_Side_Active` to be true to a "don't care."

To determine which faulty specifications contained faults that our verification suite could reveal, we reran the complete verification suite on the 100 faulty FGS

```
MACRO When_LGA_Activated() :
 TABLE
  Select_LGA()                 : T;
  PREV_STEP(..LGA) = Selected : F;
  Is_This_Side_Active          : *; /* Was T */
 END TABLE
END MACRO
```

Figure 9. An example fault seeded into the FGS model.

	Violations found	Percentage of total
Total property violations	155	
LURCH found in all 5 runs	106	68.4%
Found in at least four runs	115	74.2%
Found in at least three runs	123	79.4%
Found in at least two runs	128	82.6%
Found in at least one run	131	84.5%

Figure 10. Summary of LURCH's fault finding capability.

models using NuSMV. This extraordinarily time consuming exercise revealed that 45 of our 100 faulty models contained faults that could be revealed by our suite of properties. In addition, we found that 60 of the original 300 properties were violated in at least one of the faulty specifications. Therefore, for our experiment we selected the 45 specifications with faults we could reveal and the 60 properties that we knew were violated by at least one faulty specifications. Unfortunately, the properties are currently considered proprietary Rockwell Collins Inc. information and we can only paraphrase their informal definitions in this report. Nevertheless, the informal examples below should give the reader some understanding of the type of properties we used in this experiment.

Property 1: If the flight director cues are off, the flight director cues shall not be turned on when the Transfer Switch is pressed, (provided that no lateral or vertical mode is selected and ⟨some additional conditions⟩).

Property 2: If mode annunciation are off, auto pilot engagement shall cause `ROLL` mode to be selected (provided ⟨some additional conditions⟩).

To conduct the actual experiment we ran the verification suite on every specification using LURCH and NuSMV. Note here that we did not need to run NuSMV again to determine which properties were violated in which specification—this information was available from our initial analysis to determine which fault seeded specifications to use. The only reason to re-run the NuSMV analysis was to collect performance data using our 60 selected properties instead of the full suite of close to 300. We ran NuSMV with command options `-dynamic` (dynamic variable reordering) and `-coi` (cone of influence reduction). Without these options, NuSMV was unable to build the symbolic representation of the model. Furthermore, collecting the peak memory usage of NuSMV over very long verification runs was a nontrivial task. We ovserved memory usage manually for some runs, and it ranged from approximately 25 megabytes to hundreds of megabytes. In future studies we will modify NuSMV to allow us to collect more accurate data on it memory usage. We ran LURCH with the option to terminate a search path after two hash collisions in the state space hash table and a cutoff for the entire search of 1,800 seconds; that is, if no violation was found within 1,800 seconds, the search was terminated. Finally, due to the random nature of LURCH, we ran each verification run five times on each FGS model. In both NuSMV and LURCH, all properties were grouped together in a batch.

3.4 Results

There were two objectives with our experiment; first, to evaluate the fault finding capability of random search as compared to full verification, and second, to assess the performance of LURCH.

Regarding the *effectiveness* of LURCH, Table 10 summarizes the fault-finding capability of LURCH as compared to the results achieved with full verification in NuSMV. Each specification contained one fault, but that fault may have led to the violation of more than one property. Therefore, we report the total number of property violations over all specifications—a total of 155. The rows in the table shows the number and percentage of property violations found by LURCH. The property violations detected by LURCH identified 40 of the 45 faults seeded in our specifications (88.9%).

As can be seen in Figure 10, for some properties LURCH either failed to find a counterexample or only found it on some runs. This inaccuracy is not surprising given the incomplete searches in LURCH. In fact, some sceptical members of the research team expected LURCH to be much less accurate than what we observed in this experiment. Furthermore, by extending the search depth during "tinkering" with the search parameters of LURCH, we have been able to find additional property violations reliably[3]. Our experiences

[3]The results gathered while informally experimenting with

lead us to believe that the performance of LURCH can be improved by tuning its search parameters—in the experiment we used settings that "seemed reasonable" based on prior experience.

To better understand the characteristics of the faults that LURCH never detected, we investigated the relationship between the properties and the seeded faults that LURCH failed to detect. As one may expect, the failure of LURCH to detect a fault is because the fault affects only a small and difficult to reach portion of the state space. As an example, we found that one fault affected a portion of the FGS that is only invoked when the pilot presses the 'Transfer Switch' that transfers control of the aircraft from the pilot side FGS to the co-pilot side FGS (or vice versa). This event occurs when the 'Transfer Switch' variable changes from false to true. In the FGS, however, the various input signals are prioritized and the 'Transfer Switch' will only be considered if *none* of 12 other input signals remain false in two consecutive steps while the 'Transfer Switch' changes from false to true. Needless to say, the probability of this event occurring is astronomically small (10^{-26}) and, consequently, LURCH is highly unlikely to ever reveal this fault. This kind of fault is exactly the reason that random search methods like LURCH will never replace complete verification methods. Nevertheless, if we can better understand the nature of faults unlikely to be revealed by LURCH, we may be able to provide estimates of the number of faults remaining in a model after it has been 'LURCHed' as well as of the probability of encountering such a fault.

To illustrate the *efficiency* of the fault finding capability of LURCH, Figure 11 shows time it took for LURCH to find property violations. This graph is based on the time it took LURCH to find a property violation in one of the specifications and we are including data for every instance where LURCH found a violation—our experiments reported 600 property violations when performing five runs with LURCH. The graph shows that 386 violations (64%) were found within 120 seconds. As the graph illustrates, the vast majority of property violations are found quickly and there seems to be little hope to find additional violations by extending the search time. 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 "very difficult to find," regardless of how LURCH's adjustable parameters are set. The results for this particular case study hints that approximately 10% of the faults fall in this category,

LURCH are not in our results, since we wanted to keep the conditions unchanged for the formal experiment.

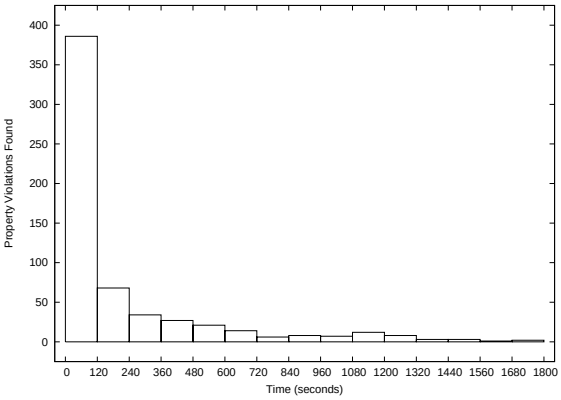


Figure 11. Number of violations LURCH found in a given time period.

but more experimentation is necessary to support or refute this claim.

To summarize, the results from our experiment indicate that techniques based on random search can find the vast majority of faults (close to 90% in this experiment) much faster (seconds and minutes as opposed to hours) and with a fraction of the memory (single digits to 10s of megabytes as opposed to 10s to 100s of megabytes) compared to a symbolic model checker. Based on these observations, we believe techniques based on random search are very effective debugging tools that should be applied to get quick results and identify faults with little effort. Furthermore, when our models become so large that formal verification becomes impossible we may have to rely on LURCH's approximate search.

4 Related Work

Any coverage of a topic area as broad as state space analysis, verification, and refutation will by necessity be incomplete. Here we only cover the work most related to the random search strategy performed in the LURCH tool.

4.1 Random Search in AI

Our experiments with LURCH have a repeated and curious feature: if an error state is reachable, then it is likely to be found quickly. Conversely, if LURCH does not reach an error state quickly, it is likely that LURCH would never find one, no matter how long it ran.

A related effect has been seen in the AI literature.

The *satisfiability* community explores models in conjunctive normal form (CNF), that is, propositional clauses such as:

$$(A \vee B \vee C) \wedge (\neg A \vee E \vee F)$$

These models combine N variables into L clauses of size K literals per clause (in our example, $K = 3$, $N = 5$, $L = 2$). Finding assignments to the variables that satisfy the clauses is a theoretically intractable task when $K = 3$. However, empirically, solving 3CNF problems is only slow in a very narrow region, when $L \approx 4.3N$ [22].

This effect is usually illustrated by something like the *phase-transition* diagram of Figure 12. In that figure, problems are placed on the x-axis according to the number of constraints per variable. When there are few constraints, problems are easy to solve, but when there are many no solution is possible. Note that in either case, it is quick to conclude that a solution exists or is impossible.

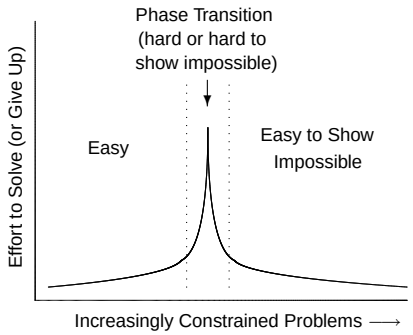


Figure 12. Hard problems exhibit a phase transition.

An often repeated empirical result for 3CNF models is that search only becomes difficult in a very narrow zone between the solvable and unsolvable problems- an effect first reported by Cheeseman [2]. This phase transition region, in the words of Cheeseman et.al., is "where the *really* hard problems are." For 3CNF, this zone is when $L \approx 4.3N$.

The phase transition effect has only been seen in artificially generated problems and not in real-world formal models like our flight guidance systems Nevertheless, the phase transition effect inspired a whole family of successful random search engines. The phase transition effect promises that most problems can be solved or found to be impossible using very simple search engines such as LURCH. In AI, since the early 1990s, this possibility has been explored by the *GSAT* family of algorithms. GSAT is a hill-climbing algorithm that starts by assigning a truth-value to every variable. At every iteration GSAT picks a variable and "flips" its value from true to false or vice versa. With good heuristics for selecting the variable to be flipped, these algorithms work amazingly well and scale to theories much larger than what can be processed by complete search [16].

4.2 Bounded Model Checking

Success in SAT solving has enabled the development of a symbolic search technique known as bounded model checking [1]. Bounded model checking uses a SAT solver to represent Boolean formulas to address the potentially exponential growth of the BDD representation in a traditional symbolic model checker such as NuSMV. Given a system model, a system property, and a pre-defined execution length, bounded model checking tries to find a witness of the negation of the given property by searching the system state space up to the predefined execution length (search depth). Bounded model checkers have been very effective in practice. Nevertheless, bounded model checking cannot be used for verification purposes since a search is limited to a depth of k steps—if no witness can be found, all we know is that there is no property violation within the predefined search depth. The random search strategy implemented in LURCH is not limited by search depth—LURCH can perform searches to an arbitrary depth—but it only samples the state spaces along these searches.

4.3 Explicit State Model Checking

SPIN, and other explicit state model checkers, suffer from the state space explosion problem. Elaborate techniques have been developed to tame the state space explosion problem during verification, for example, clustering [6], exploiting symmetries in the model [5], and semantic minimization [9]. Although these techniques are useful, they are unlikely to solve the state space problem in general.

The notion of random search has been explored to some extent in this domain. Holzmann investigated random search, called scatter search, in an early version of his SPIN tool [15], but he did not perform any studies of its effectiveness. Anecdotal evidence, however, indicated that if there was a fault in the model, it was likely to affect a large portion of the state space—an observation that coincides with our conclusions from this experiment.

Instead of exploring random search strategies within existing model checkers implemented for full verification, we have opted to implement a *simple* tool custom

made for this type of search—our current version of LURCH is less than 1000 lines of C code. By way of contrast, other researchers report that augmenting standard model checkers with heuristic search is quite difficult. For example, Edelkamp et.al. [10] report that the internals of SPIN are so complex it took nearly a year and the advice of a hard-to-reach expert before to add a simple A* heuristic search.

5 Future Work

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 inaccuracy of LURCH on realistic models is a critical issue if random search techniques 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.

It would be ideal if we could characterize what kinds of models can be adequately searched by LURCH. The phase transition effect offers us one line of inquiry. An alternate line of inquiry is the *relative size* effect. Whether a violation can be easily detected or not may be a function of the nature of the fault causing it. If the random search is viewed as a *Monte Carlo* simulation on the composite state space, the experiments are evaluating the *relative size* of the faulty state space. Further work is required to investigate this possibility.

The 45 faulty models used in our experiment were seeded with classes of faults observed in the older FGS versions. To a large degree, they reflect the errors that a human specifier tends to make when writing an RSML^{-e} model. 40 of them lead to at least one property violation that is easy to find. Although our experiment is quite extensive, we cannot conclude that most software errors affect a large portion of the state space—our experiences are limited to the FGS model and it constitutes only one data point in a wide spectrum of possible models. Clearly, we need to test LURCH on more models. To facilitate this, we are working towards a distribution of LURCH (complete with tutorial training material). We also are adding LURCH into the NIMBUS toolkit from the University of Minnesota. Both these distributions will be made freely available to researchers.[4]

[4]To access NIMBUS+LURCH, email `heimdahl@cs.umn.edu`. To download LURCH, go to `www.menzies.us/lurch.html`.

6 Conclusion

Based on our observations in this experiment, we believe techniques based on random search are very effective debugging tools that should be applied to get quick results and identify faults with little effort. The modeling activities advocated in, for example, specification centered development [3,23], SCR [13], and KAOS [24], all use model checking routinely during development. We believe tools based on random search would be a more effective fault identification tool until all but the toughest faults have been removed.

We also believe that LURCH can have an important role in supporting the full verification process. Most specifications are too large for full verification unless they are abstracted to a simpler model—hopefully an abstract model retaining all interesting behaviors. We propose to use LURCH on the full model and if LURCH's incomplete search can find faults in the original model, *while the verification on the abstraction succeeds*, then the abstraction has skipped important details. Hence, we advise always running LURCH on the current version of the original model as an additional check increasing the confidence in the verification process.

Finally, when models become so large that formal verification becomes impossible we may have to rely on LURCH's approximate search. The experiments reported here, and elsewhere [19], give us some confidence that while LURCH's search is incomplete, it can still find a majority of the faults in these larger models. With more experiments and a better understanding of how to tune the search parameters, we may even be able to provide boundaries on the expected number of remaining faults in a model searched by LURCH and possibly provide reliability predictions.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *In Proceedings of Tools and Algorithms for the Analysis and Construction of Systems*, page 193207, May 1999.

[2] 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.

[3] Y. Choi and M. Heimdahl. Model checking RSML^{-e}requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, October 2002.

[4] 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.

[5] E. Clark and T. Filkorn. Exploiting symmetry in temporal logic model checking. In *Fifth International Conference on Computer Aided Verification*. Springer-Verlag, 1993.

[6] E. Clark and D. E. Long. Compositional model checking. In *Fourth Annual Symposium on Logic in Computer Science*, 1989.

[7] E. Clarke, O. Grumberg, and D. Long. Verification Tools for Finite-State Concurrent Systems. *A Decade of Concurrency—Reflections and Perspectives*, 803, 1993.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[9] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings ICSE2000, Limerick, Ireland*, pages 439–448, 2000.

[10] S. Edelkamp, L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *Proc. AAAI Stanford Spring Symposium*, 2001.

[11] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[12] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996. Available from `http://citeseer.nj.nec.com/heitmeyer96automated.html`.

[13] C. Heitmeyer, J. K. Jr., B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[14] G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[15] G. J. Holzmann. Automated protocal alidation in argos: Assertion proving and scatter searching. *IEEE Transactions on Software Engineering*, 13(6), June 1987.

[16] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, Aug. 4–8 1996. AAAI Press / MIT Press. Available from `http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps`.

[17] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[18] T. Menzies, D. Owen, and B. Cukic. Saturation Effects in Formal Verification. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2002.

[19] T. Menzies, J. D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and D. J. When can we test less? In *IEEE Metrics'03*, 2003. Available from `http://menzies.us/pdf/03metrics.pdf`.

[20] 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.

[21] D. Owen, T. Menzies, M. Heimdahl, and J. Gao. On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate. In *IEEE NASE SEW 2003 (submitted)*, 2003. Available from `http://menzies.us/pdf/03lurchc.pdf`.

[22] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *AAAI '92*, pages 440–446, 1992.

[23] 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.

[24] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*, November 1998.

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