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

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, automated verification tools become increasingly important. Unfortunately many systems are still too large for complete verification. Here we describe LURCH, our prototype approximate verification tool, and then present experiments comparing LURCH to the popular SMV and SPIN model checkers. First, we show that for artificially generated models LURCH is able to find errors in very large state spaces using a relatively small amount of memory. Next we show that LURCH can produce surprisingly complete results, even when compared to SMV, on a large, real-world model of a flight guidance system. The cost of our technique is a slight reduction in the accuracy of our verification. However, the benefit of LURCH is scalability: this technique can find errors in models that are too big for standard model checkers.*

## 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, 8]. 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 [9].

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.

In section 2, we discuss the idea of a *phase transition*, which suggests one reason why simple, efficient strategies may be effective for many potentially very difficult problems. Section 3 describes LURCH. The following three sections compare LURCH's performance with the complete verification tools SMV and SPIN, running first on randomly generated models and then on a large, real-world flight guidance system model.

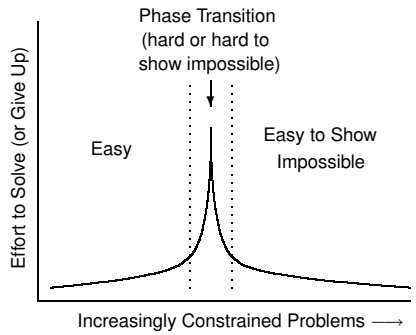


**Figure 1. Hard problems exhibit a phase transition.**

## 2 The Phase Transition

Difficult search problems, e.g., *NP-hard* problems, have been shown to exhibit a *phase transition* (figure 1) [1, 7, 13]. 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 [17]. 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.

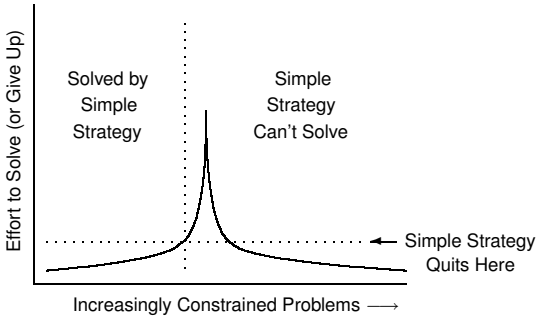


**Figure 2. The power of a simple solution strategy.**

One very simple search method are random search methods that use (1) a fast partial search, (2) a random selection amongst options, and (3) the occasional reset/restart. For example, the GSAT family of algorithms uses hill-climbing in order to test for CNF satisfiability. Given a set of propositional clauses like

$$(A \vee B \vee C) \wedge (D \vee E \vee F) \wedge \ldots$$

GSAT 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 [10].

This rest of this paper describes experiments with the LURCH random search algorithm. Which GSAT executes over CNF models, LURCH is designed for statecharts modelled as transition functions. An alternative to building a new algorithm like LURCH would be to add (e.g.) GSAT to a standard model checker. We did not use that option for *theoretical, pragmatic, historical*, and *empirical* reasons. *Theoretically*, we have some analytical results offering us some confidence in the generality of LURCH-style inference [15]. *Pragmatically*, the standard modelling tool used in the software verification community are the statecharts expressible in LURCH, not the CNF formulae required for GSAT. *Historically*, we have more experience with the AND-OR graph search used in LURCH (described in the next section) than the GSAT-style inference that uses conjunctive normal forms. Hence, we found that LURCH-style inference was very simple to implement. For example, our current version of LURCH is less than 1000 lines long. By way of contrast, other researchers report that augmenting standard model checkers with heuristic search is quite difficult[1]. Finally, *empirically*, we have results that LURCH can perform as well,

[1] For example,Edelkamp et. al. [5], report that the internals of SPIN are so complex, that it took nearly a year and the advice of a hard-to-reach expert before they could add in a simple A* search

```
1: next-global-state(state) {
2: Execute a transition for every machine in which there is
     at least one whose input conditions are satis£ed; 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-de£ned maximum reached) }
```

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

or even better, than standard model checkers such as SPIN or SMV (see below).

## 3 Approximate Verification

Model checking is used to verify that finite-state concurrent systems satisfy specified temporal logic properties [3, 8]. 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 [14, 16, 17]. LURCH uses an memory-saving AND-OR graph representation of the composite system behavior[2]. 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. The efficiency of the AND-OR graph is illustrated for a range of models representing the Dining Philosophers problem by figure 4 (we return to this problem in the next section): LURCH's memory requirements grows far slower than the exponential memory requirements of SPIN.

[2]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 [16].

The algorithm is *partial* because, unlike the full model checking technique, only a portion of possible behavior is explored; the algorithm is *random* because the choice of which behavior to explore is nondeterministic. In practice, LURCH acts as the simple solution strategy illustrated in figure 2, and, as indicated by the experiments presented below, LURCH is surprisingly successful compared to more sophisticated model checking tools.

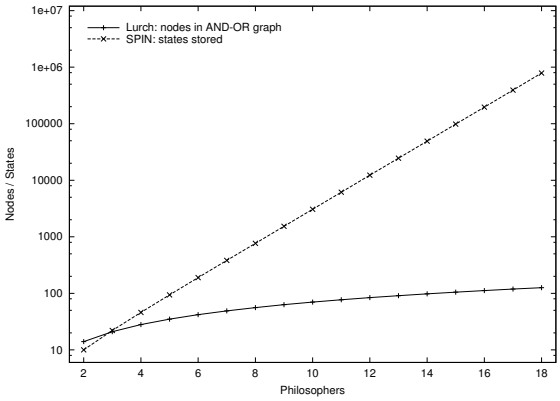


**Figure 4. For Dining Philosophers problem models, size of LURCH's AND-OR graph vs. SPIN's stored global states.**

LURCH is implemented as a *Monte Carlo* algorithm: the basic search procedure runs again and again, each time increasing the probability of finding a solution. In many cases LURCH quickly finds a solution, but for those in which LURCH does not find a solution, how do we know when to stop?

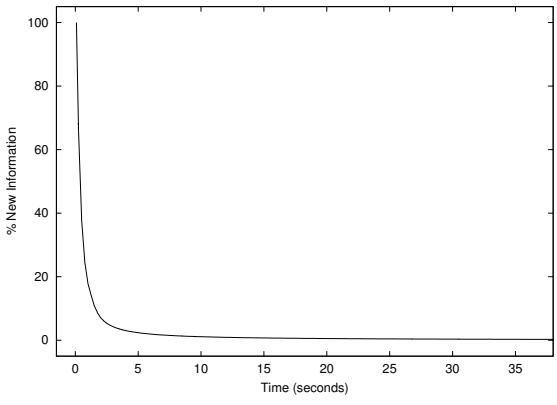


**Figure 5. LURCH output for a typical model: quick saturation.**

Figure 5 shows output from LURCH running on a typical input model. As LURCH runs, it explores the reachable global state space, at first finding nearly all new global state information, but after a little while most of LURCH's findings are redundant; figure 5 illustrates this: the percentage of global state information which is new (vs. redundant) starts out at 100 %, but very quickly decreases to near zero. We use this quick *saturation* effect in LURCH output (see [14]) to determine when to stop: when some set saturation point (close to 0 %) is reached, we assume that LURCH is unlikely to find any more interesting information.[3]

Figure 5 shows that for typical models LURCH, if it is likely to find a solution, is likely to find it quickly. Conversely, if LURCH does not find a solution quickly, it is likely that LURCH would never find a solution, no matter how long it ran. This may seem counterintuitive, saying essentially: if it's not obvious, it's not there at all; but remember figures 1 and 2: unless we were in the phase transition region, this is just what we would expect. For problem cases in the easy region, solutions are obvious. For problem cases in the region easily shown impossible, it's obvious that there is no solution.

To efficiently track which global states have been reached LURCH stores hash values based on the names of all local states present in the global state to be stored. Each global 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 global states although they are actually *potential* repeat global states. LURCH allows the user to limit the amount of memory available for global state storage.

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.

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 (attempt 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 tracked, as in an asynchronous system. We have used LURCH in asynchronous mode to find to find deadlocks in Dining Philosophers problem, as described below.

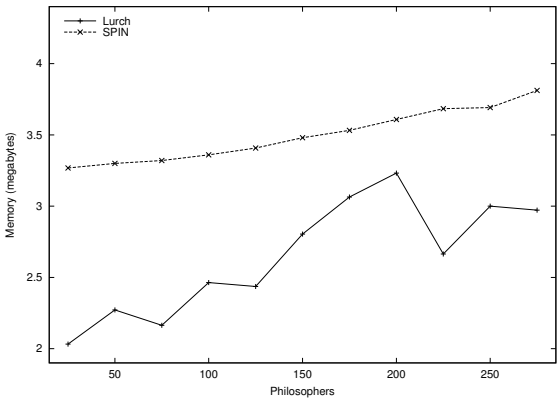


**Figure 6. LURCH and SPIN memory use for models based on the Dining Philosophers problem.**

## 4 Dining Philosophers and N-Queens Problems

Figure 6 compares memory required for LURCH and SPIN searching for deadlocks in models representing the Dining Philosophers problem, a well-known abstraction of processes competing for the use of shared resources. In this problem a number of philosophers (processes that think, wait, and eat) are seated around a table with a fork between each and their two neighbors. When a philosopher stops thinking and decides

[3]For very large input models, such as the flight guidance system described later, it is not practical to wait for global state saturation; we are continuing to experiment with other stopping criteria so that LURCH can run as quickly as possible, but with consistent results.

to eat, they pick up one fork, then another, eat, and then replace both forks. If every philosopher decides to eat at the same time, and they all pick up the left fork (or if they all pick up the right fork) a deadlock occurs: none can eat and none will allow any other to eat. For LURCH, a global state which is at the end of a path and which contains an invalid end state value for one or more local machines is considered a deadlock, e.g., a global end state is reached when all forks are in the state "held by philosopher on left" since at that point all prevent each other from moving forward; but this is not a legal end state, since all of the processes are waiting to move forward.

We have plotted memory use for models representing 25, 50, 75, ... 275 philosophers. Both LURCH and SPIN terminated as soon as an error was found; otherwise the output would likely resemble figure 4, which describes data structures used to store the entire global state space. In this experiment LURCH found deadlocks using less memory than SPIN, but for the largest models SPIN ran much faster. This may be because the models are highly symmetric, and paths to the deadlock are relatively short, i.e., they are conducive to heuristics and optimization strategies not present in LURCH.

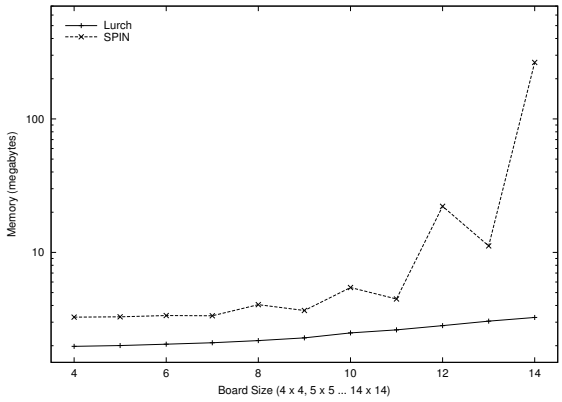


**Figure 7. LURCH and SPIN memory use for models based on the N-Queens problem.**

Figure 7 compares LURCH and SPIN memory requirements for models based on the N-Queens problem. The object is to place N queens on an N×N board in such a way that none attacks any other. Input models were written so that each tool began with a blank board and randomly assigned queens until 1) N queens were placed legally, or 2) fewer than N queens were placed but no more could legally be placed. Case 1) was described as an error state in the model, so that the tool would report it and terminate when it was discovered.

In this case LURCH is able to solve the search problem using much less memory than SPIN, which actually runs out of memory for boards 14×14 and above (note the logarithmic scale used for the y-axis). One very interesting thing about this problem: SPIN appears to require much more memory for even-numbered board sizes. This is perhaps due to the fact that SPIN's systematic search of the global state space must for some reason explore more of the space before solving the problem, for even-numbered board sizes. More importantly, this suggests one of the well-known advantages randomized strategies have over systematic strategies: they are less vulnerable to idiosyncrasies in the input. While their behavior is not predictable in a specific case, it may actually be more consistent over a range of cases.

## 5 Tic-Tac-Toe

Figure 9 shows results from a series of experiments comparing LURCH (our simple, approximate verification strategy), to two widely used model checking tools, SMV [12] and SPIN [8]. In this experiment models were generated based on a simple tic-tac-toe game. For board sizes ranging from 2×2 (worst-case composite size 162 states) through 15×15 (worst-case composite size $4.5\times10^{107}$ states), with some spaces initially assigned at random, each verification tool was used to determine whether it was still possible for either player to win (this was expressed as a temporal logic property). Figure 8 shows two examples, a board for which it is still possible for one player to win and a board for which it is not.

| | | | |
|---|---|---|---|
| X | | | O |
| | O | X | X |
| O | X | O | X |
| | | O | |

| | | | |
|---|---|---|---|
| X | | | O |
| | O | X | X |
| O | X | O | |
| | | O | X |

**Figure 8. Two tic-tac-toe boards: for the board on the left, it's still possible for someone to win; for the other it's not.**

Why tic-tac-toe? Because it's easy for a person to look at the board for a game in progress and determine whether it's still possible for one of the players to win. You would just check that all of the horizontal, vertical, and two diagonal rows contain both X's and O's; if so, obviously neither player can win. But we have generated input models so that for LURCH (or SMV or SPIN) there is no easy solution oracle. The tools are

forced to actually simulate possible sequences of play until they find a winner.

The upper plot of figure 9 begins tracking memory for 6×6 boards because models for smaller boards were verified so quickly it was difficult to get fair memory use data (memory was logged by the Windows XP *typeperf* utility). In its normal complete search mode, SPIN exceeds 250 megabytes for 8×8 boards, and continues at around 260 megabytes through board size 12×12. One convenient feature of SPIN is that it runs out of memory gracefully: there is an error message and the program terminates. So in the middle plot, from board size 8×8 to 12×12, the SPIN plot is actually showing how much memory was used before SPIN gave up; likewise, for board size 8×8 to 12×12, the SPIN plot shows how much time it took SPIN to run out of memory.

We also ran SPIN in its memory-saving *supertrace* approximation mode. This required very little memory compared to the other methods. The middle plot shows that for SMV memory spikes around 260 megabytes for

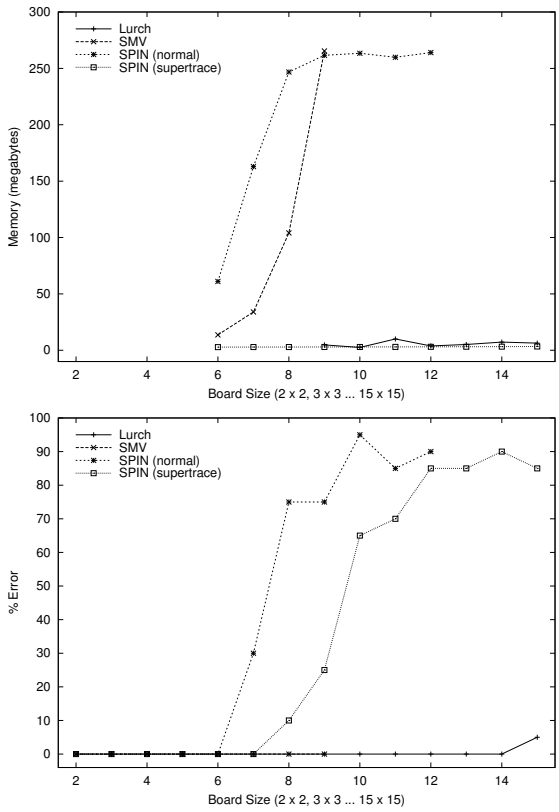


**Figure 9. LURCH, SMV and SPIN running on models based on tic-tac-toe games. Results are average values for 20 randomly generated boards of each size.**

9×9 boards. For any larger boards, SMV required a lot of virtual memory and therefore ran so slowly it was not practical to continue. LURCH, finally, required average memory between 2 and 9 megabytes for boards between 9×9 and 15×15.

The lower plot of figure 9 compares the accuracy (in terms of % error) of LURCH, SMV and SPIN in this experiment. Where the complete verification tools SMV and SPIN are inaccurate, it is because they ran out of memory before finding the error (that it was still possible for a player to win). Although it is difficult to see, SMV is accurate (0 % error) from board size 2×2 through 9×9, at which point it was no longer used because it ran out of memory for larger boards. When SPIN reaches 7×7 and begins to run out of memory for some of the runs, its % error quickly rises, so that for boards from 8×8 through 12×12 it's running out of memory on most boards before finding the error. SPIN in supertrace mode does better, but shows significant error for boards over 9×9.

Only LURCH continues to be reliable through 15×15 boards, at which point we get the first incorrect answer from LURCH; that is, LURCH terminated before finding that a player could win and reported that neither could win—when in fact it was possible for a player to win. Here we should distinguish between problem cases in the phase transition region (figures 1 and 2) and cases which are simply too large to be solved easily. The key difference is that cases in the phase transition are significantly more difficult than other problems of the same size in the easy and easily-shown-impossible regions. With this in mind, LURCH's failure here could be explained in at least two ways. First, we may be approaching the point where the problems are too large, whether or not they are in the phase transition. Second (and more likely), it could be that we have been solving phase transition problems every so often throughout the experiment and only at 15×15 are the problems big enough to thwart LURCH's simple solution strategy. In any case, the fact that LURCH outperforms more sophisticated strategies suggests that no more than a few of the problem cases encountered could have been the pathological phase transition cases.

## 6 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 developed in collaboration between Rockwell Collins Inc. and the University of Minnesota. The mode logic is captured in RSML$^{-e}$

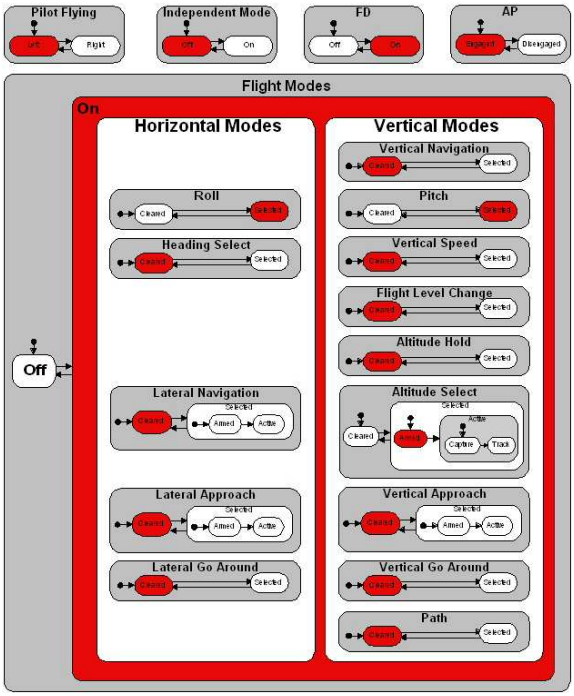


**Figure 10. Flight Guidance System**

and automatically translated to SMV and LURCH through NIMBUS, the development environment for RSML$^{-e}$ [18–20].

## 6.1 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 down to mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. In this case study we have used the mode logic.

Figure 10 illustrates a graphical view of a 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.

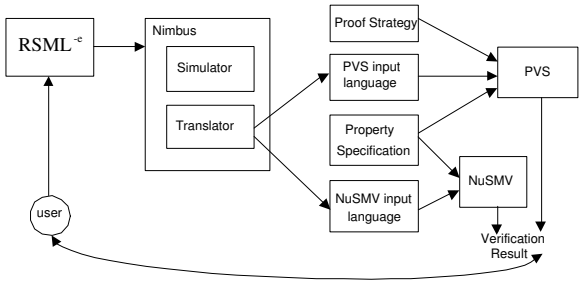


**Figure 11. Veri£cation Framework.**

## 6.2 NIMBUS and RSML$^{-e}$

Figure 11 shows an overview of the NIMBUS tools framework. The user builds a behavioral model of the system in the fully formal and executable specification language RSML$^{-e}$ (see below). After evaluating the functionality and behavioral correctness of the specification using the NIMBUS simulator, users can translate the specifications to the PVS, NuSMV, or LURCH input languages.

RSML$^{-e}$ is based on the statecharts [6] like language Requirements State Machine Language (RSML) [11]. RSML$^{-e}$ is a fully formal and synchronous data-flow 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 providing increased readability and ease of use.

Figure 12 shows a specification fragment of an RSML$^{-e}$ specification of the Flight Guidance System[4]. 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 in the definition. The condition tables are encoded in the macros `Select_ROLL` and `Deselect_ROLL`. The use of macros not only improves the readability of the

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

```
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 12. A small portion of the FGS speci£-cation in RSML$^{-e}$ .**

specifications but also helps to localize errors and future changes. The tables are adopted from the original RSML notation–each column of truth values represents a conjunction of the propositions in the leftmost column (a '*' 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 a disjunctive normal form.

### 6.3 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. As an example, figure 13 shows a missing condition fault contained in macro `When_LGA_Activated`, the fault was created by

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

**Figure 13. An example fault seeded into FGS model.**

commenting out the condition `Is_This_Side_Active = True`.

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. The properties are 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 property we used in this experiment.

**Property 4:** 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 8:** If mode annunciations are off, auto pilot engagement shall cause ROLL mode to be selected (provided ⟨some additional conditions⟩).

### 6.4 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)

| 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.**

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[5]. 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 default 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.

## 7 Conclusion

LURCH is an attempt to extend some of the benefits of model checking to systems too large or irregularly structured for complete verification tools like SMV or SPIN. The *phase transition* affect observed by others suggests that exhaustive search strategies may be overkill for many complex problems. If this is true, a simple, efficient (albeit incomplete) strategy like LURCH may be an effective way to check for many of the errors possible in systems too large for complete verification.

Our experiments with set of randomly generated input models show that LURCH is able to scale at least as well, and often much better than standard model checking tools, to large state spaces. LURCH found safety and liveness violations in these randomly generated input models using in some cases orders of magnitude less memory. More importantly, LURCH found errors in models so large that SMV and SPIN ran out of memory before complete verification could be accomplished.

One interesting result (recall figure 7 and the N-Queens problem) suggests that LURCH is less suscep-

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

tible than SPIN to idiosyncrasies in the input model. This is one of the advantages of randomized algorithms recognized in other domains: they are less likely to be biased in favor of or against a particular problem case. Continuing experiments with LURCH should help us understand whether this is a significant advantage for model checking problems.

Finally, the flight guidance system experiment summarized above shows that LURCH is (at the very least) a promising addition to the range of available model checking tools. Results were surprisingly complete and consistent, even for a very large and complex model. Our future work will focus on exploring how LURCH's adjustable parameters should be tuned to different types of models. We will also continue to explore what makes LURCH work better with some types of models than with others.

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

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

[7] B. Hayes. On the Threshold. *American Scientist*, 91(1), 2003.

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

[9] 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.

[10] 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`.

[11] 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.

[12] K. L. McMillan. The SMV System, 2000. Available at `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.

[13] T. Menzies and B. Cukic. Adequacy of Limited Testing for Knowledge-Based Systems. *International Journal on Artificial Intelligence Tools*, 9(1), 2000.

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

[15] T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In M. Madravio, editor, *Soft Computing in Software Engineering*. Springer-Verlag, 2003. Available from `http://menzies.us/pdf/03maybe.pdf`.

[16] 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.

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

[18] 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.

[19] 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.

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