

# A Shot in the Dark

David Owen  
Lane Department of Computer Science  
and Electrical Engineering  
West Virginia University  
Morgantown, WV 26506  
drobo75@hotmail.com

## Abstract

*What can we learn from a shot in the dark? What can a relatively simple, random procedure tell us about an unknown and potentially complex space? Simple techniques based on randomized algorithms have been surprisingly successful in solving very difficult computational problems. This may be due to a phase transition observed for certain problems: most problem cases are easy to solve or easily shown unsolvable; few approach the worst case. Or it may be because of funnels: small sets of key variables, found in many systems, that largely determine the behavior of the entire system.*

*We summarize experiments motivated by these ideas, experiments in which 1) a simple random search outperformed more sophisticated strategies in finding subtle errors in software models, and 2) machine learning was used to determine which models are best and worst candidates for random search. We conclude with a broader application of these ideas, suggesting that the success of random search procedures, sometimes cited as evidence that intelligent design is unnecessary, may actually be evidence of intelligent design present in the search space.*

## 1 Introduction

What can we learn from a shot in the dark? Can we hope to hit something that will tell us what the darkness conceals? Or, if we know something about what the darkness conceals—if, for example, we are shooting from the side of a ship on a cloudy night—what would we expect to hit? Probably the shot would be followed by the sound of water splashing in the distance. On the other hand, if we know nothing about the surroundings but after firing hear the sound of a splash, we would

be right to assume that there is a large body of water not far away.

To reformulate this metaphor in terms of a search problem in computer science: what can a relatively simple, random procedure tell us about an unknown and potentially very complex space of possibilities? Suppose we do know something about the structure of the space; based on what we know, what do we expect to learn from a simple, random (and not necessarily exhaustive) exploration procedure? Or, if that procedure leads us quickly to interesting places in the space of possibilities, places thought to be very difficult to reach, what does that tell us about the actual structure of the search space?

These are some of the interesting questions related to and perhaps partially answered by the research summarized in this paper. Most of the examples come from cooperative work between NASA and West Virginia University, where randomized algorithms are applied to a software model verification technique called *temporal logic model checking*.

Model checking is used to automatically verify that the overall behavior of a system of interacting parts satisfies specified properties. Over the last twenty years, automatic verification by model checking has been effective in many areas, including computer hardware design, networking, security and telecommunications protocols, automated control systems and others [2, 4, 8].

Unfortunately, the space of possible behaviors associated with models of concurrent software systems quickly grows very large and very complex, compared to the size of the models. This is because a description of the overall behavior must include every possible interleaving of the behaviors of the individual parts. For many models the overall space of behaviors is so large that verification tools designed to systematically explore the entire space, e.g., model checkers, require prohibitively large amounts of memory and

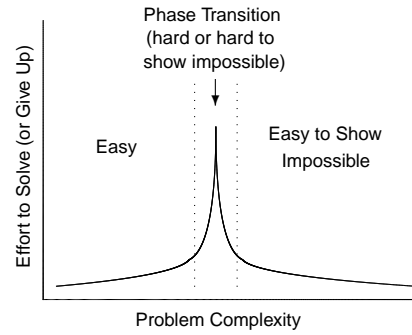
time. An approximate verification strategy based on a randomized search procedure, however, need not construct and keep track of the entire space of possible behaviors. The randomized software verification strategy described below simply explores one path through the space—one branch in a tree of possible behaviors—at a time. If the path leads to something significant, it is reported. Otherwise, the search is repeated until it finds something significant, or until a set time limit is reached.

Sections 2–4 cover some of the background research and ideas motivating the use of relatively simple random search techniques on potentially very large and complex problems. Section 5 describes briefly how model checking is used to verify finite-state models representing software systems and then describes Lurch, a random search model checker in development at West Virginia University. Section 6 summarizes one experiment in which Lurch outperformed two popular exhaustive search model checkers for very large models. Section 7 shows one method of tracking the progress of a random search and attempts to answer the question: how do we know when to stop searching? Sections 8 and 9 explain how a tool like Lurch actually simulates emergent behavior of a system’s individual parts interacting through time (and how that is useful in practice). Section 10 summarizes an experiment in which a machine learning tool was used to determine what makes certain models amenable to verification by random search. The final section suggests how some of the ideas associated with this research might help answer much broader questions about the relationship between design and emergent behavior.

## 2 Rooms and Caves

Suppose a man is kidnapped, blindfolded and left in a dark, unfamiliar room. He hears his captors leave and tears off his blindfold, but the room is completely dark. Curious about his surroundings, he stands up and extends his arms, moving slowly forward until he reaches one of the walls. He follows that wall with his hands to a corner of the room and then to a door frame. As he tries to open the door, his elbow bumps into a light switch, turning it on. He sees that the room is small, rectangular and empty.

Now suppose the same man is kidnapped, blindfolded and thrown into a dark, unfamiliar *cave*. He tears off his blindfold, but it’s completely dark; he still can’t see anything. He stumbles forward hoping to determine the shape of the cave. He can feel that the floor and walls are brittle and uneven, there are loose rocks and puddles everywhere, bats chasing insects fly-



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

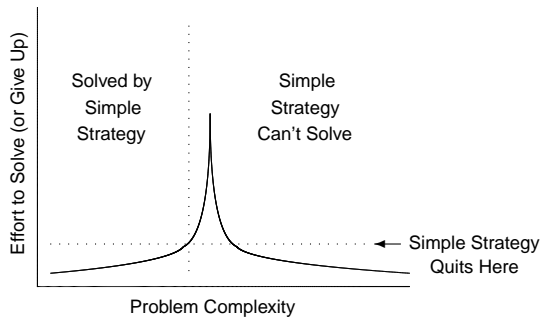
ing around his head—so he just starts crawling around on his hands and knees thinking: maybe there’s a way out? Maybe someone left a match on the floor?

In a small, rectangular, empty room, a very simple strategy can quickly determine the shape of the room. Like the captive man described above, you could just slide your hands along the wall until you bump into a light switch. A more sophisticated strategy could be used; for example, to keep track of exactly what portion of the room you had already explored, you might take a key from your pocket and carefully gouge marks on the floor every inch or so, going in a straight line until you reach a wall, and then turning around and marking out another line next to the first in the opposite direction. Clearly this would be a waste of time. On the other hand, in the cave a simple strategy is practically hopeless, and even a very sophisticated strategy may not work, because the problem is so difficult.

## 3 The Phase Transition

This problem of finding your way in a dark, unfamiliar room is somewhat analogous to a class of (potentially) very difficult problems in computer science,<sup>1</sup> a class of problems said to exhibit a *phase transition* (figure 1). In some cases, like the simple empty room above, the problem turns out to be very easy to solve. Other cases, like the cave, are practically impossible. For most of these impossible cases, however, it is easy to show even with a very simple strategy that they can not be solved (after tripping over a few rocks on the

<sup>1</sup>In precise terms, this is the class of *NP-hard* problems, generally considered intractable: it is theoretically possible to find a solution, if one exists, but in the worst-case analysis this will require prohibitively large amounts of time and memory. On the other hand, a potential solution can be checked efficiently. For more on the class *NP* see, e.g., [19].



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

floor of the cave, you realize it’s a cave and therefore has no light switch).

So there are easy cases and cases easily 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 some researchers, is “where the *really* hard problems are” [1, 7, 11].

Figure 2 shows how a simple solution strategy can be used to exploit easy problem cases but avoid wasting effort on cases that are very hard or unsolvable [18]. 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. In practice a simple strategy is often more effective because its efficiency allows it to scale to much larger problem instances, as will be shown below.

## 4 Funnels

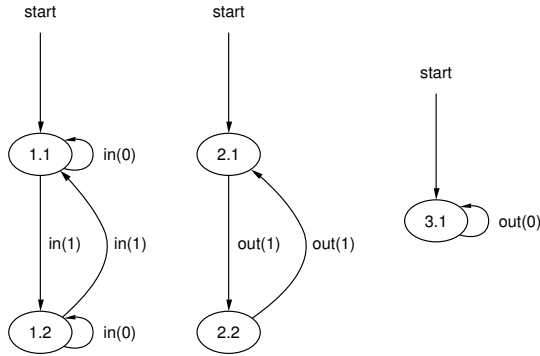
Motivated by results like the *phase transition* described above, software engineering researchers Menzies and Cukic have argued that “standard models of test suite sizes are gross overestimates” because they do not consider what we know about the structure of typical computer programs [10]. In a related work they

cite examples of an apparently very complex natural language processing system and a large study of thousands of Fortran and C programs; in all cases program structure turned out to be surprisingly simple [12]. Ironically, adequate testing is not done because developers assume it would be too expensive. Simpler, cheaper approaches that take advantage of what we know about typical software systems—that the general verification problem might follow the pattern of a phase transition—may help improve software development in the future.

The *funnel theory* of Menzies et.al. summarizes and attempts to explain why simple strategies have been surprisingly effective in solving apparently difficult computational problems, why results from research on different kinds of systems suggest that many fall in the easily solvable range of the phase transition plots above. It seems that these systems contain *funnels*: small sets of key variables that determine the behavior of everything else [15]. The key variables form a virtual funnel in the structure of the space of possible behaviors; a large proportion of the possible search paths are forced to go through the small space of behaviors represented by the funnel. A simple search strategy quickly finds the funnel, because so many possible search paths lead to it. And a more systematic search strategy yields little (if any) new information because all of the paths it systematically checks lead to the same funnel. Menzies and Singh go further to show mathematically that, where funnels are present, random search will with high probability find the most *narrow* funnels—that is, the smallest sets of key variables [15]. This is an ideal result for testing applications: given some interesting behavior, we would like to know the most concise explanation for it.

## 5 A Simple Strategy for Software Verification

The research and experiments referred to in this paper are primarily concerned with the task of automatically verifying that a software model satisfies a formal specification. Although the general problem of software verification can not be solved by a computer [19], a technique called *model checking* is sometimes used to verify that an abstraction of the program satisfies specified properties [3, 8]. For example, figure 3 shows a simple program model and a property for which the model might be verified. The leftmost component of the model represents a very simple computer program, and the other two components represent sources of data input by the program. The program modeled here tracks whether an odd or even number of 1’s is input



**Figure 3. A simple program model satisfying the property:  $\text{always}(2.2 \rightarrow \text{eventually}(1.2))$**

but ignores 0's.

The form of the model in figure 3 is called a *finite-state concurrent system*, because it consists of several concurrently executing parts (or *machines*), each with a finite number of states (and a finite number of transitions between states). The *always* and *eventually* operators come from *temporal logic*, the formal mathematical language used to specify properties in model checking. For example, while Boolean logic could be used to state the property that  $A$  implies  $B$ —if  $A$  is true then  $B$  is also true—temporal logic goes further, allowing such constructions as:  $A$  implies *eventually*  $B$ , which means that if  $A$  is initially true  $B$  is true now or will be at some point in the future. Temporal logic makes it possible to specify many interesting and important features of real software systems.

Each of the three components in figure 3 is a finite-state machine. In this model transitions between states are capable of inputting or outputting 1's and 0's. The temporal logic property at the bottom of figure 3 states that any time finite-state machine 2 is in state 2.2, machine 1 must be in state 1.2 at that time or at some point in the future; less formally, if at any time an odd number of 1's have been output by machine 2 (it is in state 2.2), machine 1 will eventually input an odd number of 1's, which will put it in state 1.2. This would be true regardless of the behavior of machine 3.

The model checking technique is first to build a *composite* finite-state machine representing the entire behavior of the individual machines in the original model (all possible interleavings of their parallel operation) and then to systematically search through the entire behavior of the composite machine for violations of the property specification. The amount of memory required to store the composite machine is, in the worst case, an exponential function of the size of the original

model [4]. For example, a model with six ten-state machines could require a composite with  $10^6 = 1,000,000$  states to represent all possible behavior. Since the model checking technique requires that the composite machine be searched completely for property violations, an exponential amount of time may also be needed. For many large systems, this turns out to be a prohibitively large amount of memory and time.

The full model checking technique may be overkill, however, for software verification problems that turn out to be easy—to use the full model checking technique for these problems would be something like the illustration in section 2: carefully marking the floor as you try to find your way in an empty rectangular room. Elsewhere we have described a simple randomized-search alternative to model checking, implemented in a tool called Lurch [18]. Lurch uses a memory-saving AND-OR graph representation of the composite system behavior and a fast partial random search algorithm [14, 17].<sup>2</sup> 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 in the next section, Lurch is surprisingly successful compared to more sophisticated model checking tools.

To avoid confusion about our use of words like *random*, *randomized*, and *nondeterministic*, we put forth the following informal definition of randomness: a process is random, from some point of view, if from that point of view its behavior is not entirely predictable. This says nothing about whether the process is in any absolute sense random and (hopefully) avoids all of the difficult philosophical questions involved in such a strong claim. Nondeterministic choices in Lurch, for example, are actually based on the pseudorandom integer generator available in the C language, seeded with a value from the computer's hardware clock.

## 6 Tic-Tac-Toe

Here we summarize results from a series of experiments comparing Lurch (our simple, approximate verification strategy), to two widely used exhaustive search model checking tools, SPIN [8] (running in normal mode and its *supertrace* memory-saving approximation mode) and SMV [9]. In this experiment models were

<sup>2</sup>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 an *NP-hard* problem [17].

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

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

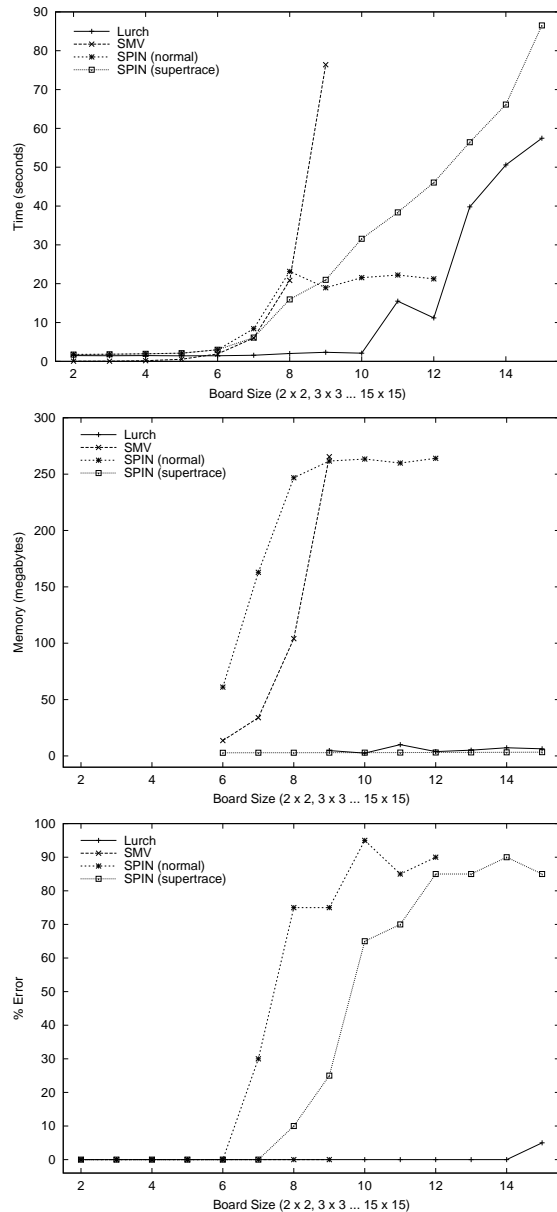
**Figure 4. Two tic-tac-toe boards: for the board on the left, it's still possible for O to win; for the other board it's no longer possible for either player to win.**

generated based on a simple tic-tac-toe game. For board sizes ranging from 2x2 (worst-case composite size 162 states) through 15x15 (worst-case composite size  $4.5 \times 10^{107}$  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 4 shows two examples, a board for which it is still possible for one player to win and a board for which it is 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 for Lurch (or SMV or SPIN) there is no easy solution oracle. The input models have been written so that the tools must actually simulate possible sequences of play until they find a winner.

In the top plot of figure 5, time for SMV spikes over one minute for 9x9 boards; SPIN (in normal mode) reaches about 20 seconds for 8x8 boards and remains there for boards up to 12x12. SPIN in supertrace mode requires about the same amount of time as SPIN in normal mode, up to 9x9 boards, and then continues to increase, reaching nearly 90 seconds for 15x15 boards. Lurch remains very fast for boards up to 10x10, and then reaches about one minute for 15x15 boards. Although it may be difficult to see on the graph, Lurch is fastest for all runs in which any technique took more than 5 seconds.

The middle plot of figure 5 begins tracking memory for 6x6 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). Here SPIN (in normal mode) reaches past 250 megabytes for 8x8 boards, and continues at around 260 megabytes through board size 12x12. One convenient feature of SPIN is that it runs out of memory gracefully: there is an error message and the program



**Figure 5. Time, memory, and accuracy comparison for Lurch, SPIN and SMV running on models based on tic-tac-toe games (average values for 20 boards of each size).**

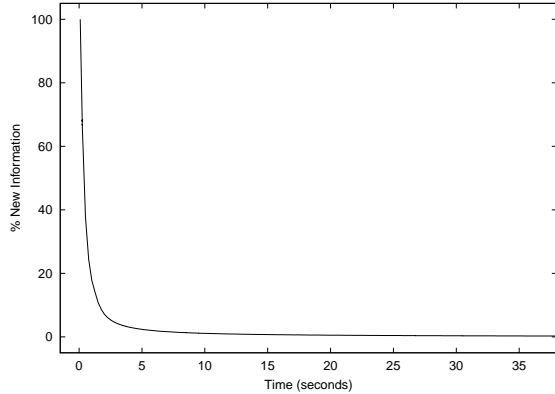
terminates. So in the middle plot, from board size  $8 \times 8$  to  $12 \times 12$ , the SPIN plot is actually showing how much memory was used before SPIN gave up; likewise, in the top plot, for board size  $8 \times 8$  to  $12 \times 12$ , the SPIN plot shows how much time it took SPIN to run out of memory. SMV, at least in the close-to-default mode we used for this experiment, did not run out of memory so gracefully. The middle plot shows a spike around 260 megabytes for  $9 \times 9$  boards. For any larger boards, SMV required a lot of virtual memory and therefore ran so slowly it was not practical to continue. The bottom right section of the middle plot shows that Lurch uses about the same amount of memory as SPIN running in its memory-saving supertrace mode, between 10 and 20 megabytes for boards up to  $15 \times 15$ .

The bottom plot of figure 5 compares the accuracy (in terms of % error) of Lurch, SMV and SPIN in this experiment.<sup>3</sup> Although it is difficult to see, SMV is accurate (0 % error) from board size  $2 \times 2$  through  $9 \times 9$ , at which point it was no longer used because it ran out of memory for larger boards. When SPIN (in normal mode) reaches  $7 \times 7$  and begins to run out of memory for some of the runs, its % error quickly rises, so that for boards from  $8 \times 8$  through  $12 \times 12$ , it is not very reliable. SPIN in supertrace mode behaves similarly, but is somewhat reliable up to boards of size  $10 \times 10$ .

Only Lurch continues to be reliable through  $15 \times 15$  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 (see 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 board size in the easy and easily-shown-impossible regions.

Lurch’s failure here could be explained in (at least) two ways. First, we may be approaching the point where the problem cases 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, but only at  $15 \times 15$  are the problems big enough to thwart Lurch’s simple solution strategy. In any case, the fact that Lurch outperforms more sophis-

<sup>3</sup>It may seem inappropriate to use the term *error* in this context, since exhaustive search model checkers like SPIN and SMV guarantee correct output—all possible behaviors are checked, so there is no possibility of error. But there is no guarantee for these tools if they run out of memory and terminate early. If that happens, SPIN and SMV, like Lurch, may fail to check significant behaviors of the system. Here we consider these failures errors.



**Figure 6. Lurch output for a typical model: quick saturation.**

ticated strategies suggests that few of the problem cases encountered have been the pathological phase transition cases; most cases have been easily solved or not solvable. For a more detailed description of this tic-tac-toe experiment, see [18].

## 7 Saturation

Lurch is implemented using a type of randomized algorithm called a *Monte Carlo* algorithm: the basic search procedure runs again and again, each time increasing the probability of finding a solution [16]. 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 6 shows output from Lurch running on a typical model, in terms of the percentage of information found by Lurch which is new (not redundant).<sup>4</sup> As Lurch runs, it explores the reachable composite state space, at first finding nearly all new information, but after a little while most of Lurch’s findings are redundant; figure 6 illustrates this: the percentage of 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

<sup>4</sup>A careful reader may realize that, in order to determine whether information is new or redundant, Lurch must have some way of tracking where it’s been. But this would require the exponential memory needed by a conventional exhaustive search model checker! Actually, Lurch does keep track of what portion of the space has been explored, but uses a memory-efficient (and lossy) hashing scheme to do it. This does not affect Lurch’s performance, however; it is only used to get a rough idea of saturation.



**Figure 7. Tree shapes representing easy (left) and hard (right) problem structures.**

point (close to 0 %) is reached, we assume that Lurch is unlikely to find any more interesting information.

Figure 6 shows clearly that for typical models Lurch, if it is likely to find some particular behavior, is likely to find it quickly. Conversely, if Lurch does not find a particular behavior quickly, it is likely that Lurch would never find it, 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.

## 8 Ants and Trees

At this point it makes sense to update the room-or-cave metaphor used in the introduction. The model checking problem solved by Lurch is more general than that of an individual trying to determine the shape of a dark room (or cave). Lurch actually simulates the emergent behavior of a system's individual parts interacting through time. The right metaphor would perhaps be: a group of people working together to track how the shape of a room can change over time—the people are the finite-state machines in the input model, and the (changing) room is a description of all their possible interacting behavior. Figure 7 gives us a less confusing way of understanding the problem. We replace the small, rectangular, empty room with a palm tree, representing simple problem structures; we replace the cave with a lower, more dense tree, representing more complex problem structures. And instead of an individual we picture cooperative ants helping (or perhaps sometimes hindering) each other climb up the tree.<sup>5</sup>

<sup>5</sup>The choice of *ants* is not entirely coincidental: we hope to use Lurch in a future NASA project involving *ANTS*—an Autonomous Nano-Technology Swarm of very small cooperating spacecraft that may eventually be used to gather information about the asteroid belt [5].

What is the difference between the structure of easy problems and very difficult or impossible problems? We suggest that it's something like the difference between the two trees pictured in figure 7. Easy problem structures constrain the search for solutions such that there is a high probability of finding the same thing over and over, like the palm tree on the left. No matter how the climbing ants interact, if they make any progress it will be toward the same destination at the top of the tree. On the other hand, some structures offer almost no guidance toward any particular destination, like the low, dense tree on the right. Here the climbing ants may become hopelessly lost, and if they were to start over they would almost certainly not reach the outcome of their previous climb.

## 9 Paths to Properties

This section summarizes how Lurch's exploration of the emergent behavior of a software model's components interacting through time is used to find subtle errors. The output of Lurch's basic random search algorithm is a single path through the composite system representing the overall behavior of the interacting finite-state machines described in the input model. Each step in the output path contains a local state value for each of the finite-state machines in the input; and consecutive steps differ by at most one transition in each machine. For example, if we think in terms of the illustration in the previous section, the composite system is the tree, and each individual finite-state machine in the input is an ant. Each step in the output path contains a single location for each ant, and in consecutive steps each ant may move only a small amount forward (ants may not instantaneously jump from one part of the tree to some distant part).

To provide the functionality of complete-search model checking tools, Lurch must 1) be able to report when particular steps take place, and 2) be able to track cycles, i.e., looping paths. For example, we may want to prove that two processes in a software system can't both access the same shared resource at the same time. If Lurch can find a path to a step in which both processes' states indicate that they are cleared to use the same shared resource, this path is a counterexample which disproves the desired *mutual exclusion* property. Or suppose we want to prove that some desired output step will eventually be reached. If Lurch can find a cycle that does not include the desired output step, this means that it is possible for the system to continue indefinitely without ever reaching the desired output step.

Figure 8 shows Lurch output for a specific example

| time | global<br>% new | local<br>states | local<br>trans. |                 |
|------|-----------------|-----------------|-----------------|-----------------|
| 0.03 | 37              | 19              | 20              | cycle w/o '+' 1 |
| 0.04 | 36              | 19              | 20              | cycle w/o '+' 2 |
| 0.04 | 35              | 19              | 20              | escaped cycle 1 |
| 0.06 | 10              | 19              | 20              | cycle w/o '+' 3 |
| 0.06 | 8               | 19              | 20              | escaped cycle 3 |
| 0.13 | 3               | 19              | 20              | cycle w/o '+' 4 |
| 0.13 | 3               | 19              | 20              | escaped cycle 4 |
| 0.13 | 3               | 19              | 20              | cycle w/o '+' 5 |
| 0.21 | 2               | 19              | 21              | escaped cycle 5 |
| 0.26 | 1               | 19              | 21              | cycle w/o '+' 6 |
| 0.30 | 1               | 19              | 21              | escaped cycle 6 |
| 0.35 | 1               | 19              | 21              | escaped cycle 2 |
| 0.36 | 1               | 19              | 21              | cycle w/o '+' 7 |
| 0.40 | 1               | 19              | 21              | cycle w/o '+' 8 |
| 1.75 | 0               | 19              | 21              |                 |

max depth: 195

**Figure 8. Lurch output for process scheduling input model.**

in which cycles were found disproving a desired property. Here the input model is a process scheduling algorithm from [8]. The purpose of the system represented by the input model is to schedule processes' use of a shared resource such that local *progress* states (represented by a '+' in Lurch syntax) are reached infinitely often. That is, it should be impossible to get stuck in a cycle that has no steps including local progress states—this undesirable situation is sometimes called a *livelock*, because although some processing is being done (the system is *live*) for all practical purposes the system is locked (no progress can be made).

In Figure 8, the left column shows Lurch's execution time in seconds. The next column shows a percentage associated with saturation (see section 7), to give an idea of how Lurch is progressing relative to the reachable composite state space. The third and fourth columns show how many of the local finite-state machines' states and transitions have been executed. The final column shows that Lurch found eight cycles, six from which it was possible to escape and two genuine livelocks. Note that the saturation (% new) value reached zero before Lurch terminated, from which we infer that Lurch found very nearly all it would ever find given infinite time. Also, the maximum search depth (the maximum steps in an output path) was 195.

## 10 What Makes a Model Easy to Search?

Here we present the results of an experiment set up to answer the question: is it possible to design software so that it will be easy to test? In other words, can we design the individual components of a system so

that the space of resulting emergent behavior is easy to explore quickly, even with a simple, random search strategy like Lurch?

For the experiments presented here, originally reported in [17], a large set of input models was generated randomly according to distributions representing typical software models. Several rules were imposed on the randomly generated models to make sure their behavior would be similar to real models. For example, each individual finite-state machine within a model had to have at least two mutually exclusive states. Also, no transition could require as an input condition two mutually exclusive states from the same machine. Attributes of each randomly generated model were recorded, as were results from Lurch running on that model, and each model was classified according to how well Lurch was able to explore the space represented by the model. A good classification indicated that Lurch quickly reached a large proportion of local states and executed most or all of the transitions in the individual finite-state machines in the model. A bad classification indicated that Lurch, even after running for a significant amount of time, could reach only a small proportion of states or execute only a small number of transitions.

A freely available machine learning tool called TAR2 [13] was used to summarize the data. A typical machine learner, given a training set of data associating attributes with classes, outputs rules to predict which class a particular case will fall into, according to that case's attributes. A *treatment learner*, TAR2 provides more concise information about the input data:

- TAR2 considers classes ordered by *score*; classes with a high score are considered better than classes with a low score, and the most desirable class (which has the highest score) is called the *best* class.
- TAR2 outputs *treatments* rather than classification rules; a treatment is an attribute range (or a conjunction of attribute ranges) that can be used as a constraint on future input cases—a guide for creating an input set of cases that fall into better classes.

Figure 9 shows a small training set with four attributes (outlook, temperature, humidity, wind) and three classes (none, some, lots). TAR2's best treatment from this data was *outlook = overcast*—based on this data, if it is overcast, most likely *lots* of golf will be played, regardless of temperature, humidity and wind conditions. If the class order is reversed (*lots* is considered the worst class and *none* the best), TAR2 can be used to find the worst treatment. Based on this

| Case | Outlook  | Attributes |          |       | Class |
|------|----------|------------|----------|-------|-------|
|      |          | Temp.      | Humidity | Wind  |       |
| 1    | sunny    | 85         | 86       | false | none  |
| 2    | sunny    | 80         | 90       | true  | none  |
| 3    | sunny    | 72         | 95       | false | none  |
| 4    | rainy    | 65         | 70       | true  | none  |
| 5    | rainy    | 71         | 96       | true  | none  |
| 6    | rainy    | 70         | 96       | false | some  |
| 7    | rainy    | 68         | 80       | false | some  |
| 8    | rainy    | 75         | 80       | false | some  |
| 9    | sunny    | 69         | 70       | false | lots  |
| 10   | sunny    | 75         | 70       | true  | lots  |
| 11   | overcast | 83         | 88       | false | lots  |
| 12   | overcast | 64         | 65       | true  | lots  |
| 13   | overcast | 72         | 90       | true  | lots  |
| 14   | overcast | 81         | 75       | false | lots  |

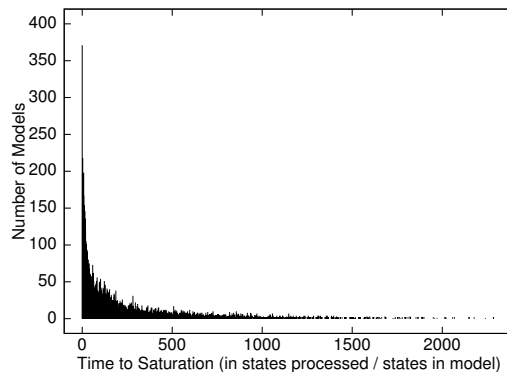
**Figure 9. A simple training set for a treatment learner (a log of golf-playing behavior).**

data, if the humidity exceeds ninety percent, very little golf will be played. To make sure these results are correct, the next step would be to check them on an independent data set. This is the usual caveat with any machine learning tool: the results are only as good as the training data.

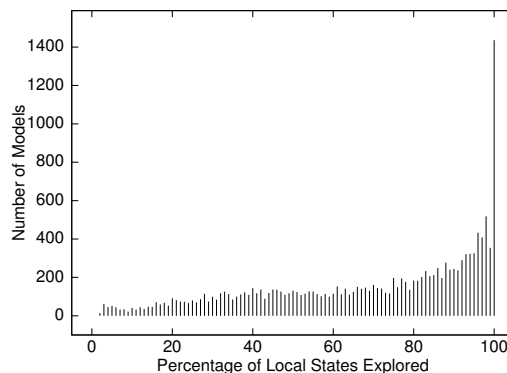
To explore the relationship between input model attributes and Lurch’s ability to explore the models, 15,000 models were generated, representing a wide range of size and density of interconnection. Figure 10 shows a summary of search results for these randomly generated models. The top histogram summarizes *time-to-saturation* results normalized by input model size. For example, for approximately 375 models saturation was achieved almost immediately, so that  $time\text{-}to\text{-}saturation = (size\ processed)/(model\ size) < 1$ . The average value was about  $208 \times (model\ size)$ . The right side of the plot shows that, for a few models, nearly  $2,500 \times$  the size of the model was processed before saturation. The bottom part of Figure 10 is a histogram summarizing the percentage of each model explored for the 15,000 randomly generated models. The average value was about 70%, with a significant number of models showing much lower values.

The top part of Figure 10 indicates that saturation was reached very quickly for nearly all models.<sup>6</sup> So the key distinction, as far as Lurch’s ability to search a particular model, is the portion of the model explored before saturation. We would like to know how models that tend to reveal information are different from models that tend to conceal information. Specifically, what ranges of input model attributes characterize models

<sup>6</sup>A factor of 2,500 is not significant, compared to the worst-case exponential size of the overall space of behaviors that would be explored by a conventional model checking tool.



Average time-to-saturation =  $208 \times$  model size.

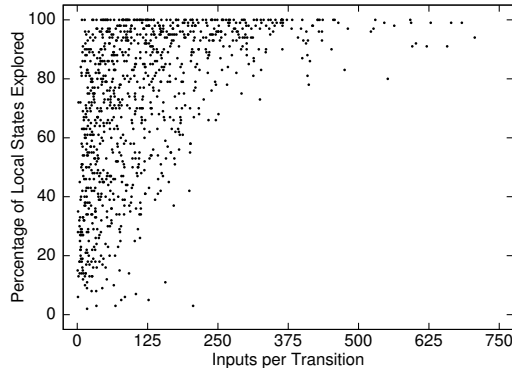


Average portion of model explored at saturation = 69.4%.

**Figure 10. Summary of time-to-saturation (top) and % explored (bottom) results for 15,000 randomly generated models.**

amenable to random search, models represented by the right side of the bottom histogram in Figure 10?

In our first simple experiment we used TAR2 to determine what single attribute, and what range of that attribute, could most significantly constrain our model set to those for which Lurch was able to explore a high percentage of the model before saturation—like the very simple golf example above, where we found that restricting *outlook* to *overcast* led to *lots* of golf. TAR2 suggested the following treatment: restrict *inputs per transition* to its highest range. To understand what that means, consider Figure 11, which shows the number of inputs per transition vs. the percentage of the model explored at saturation (there is a dot for each model). On the left, where there are few inputs per transition, we see % explored distributed all the way from about 5% to 100%. But further to the right, where the number of inputs per transition is high, we see only high values for % explored—when the num-



**Figure 11. Inputs per transition vs. % explored at saturation.**

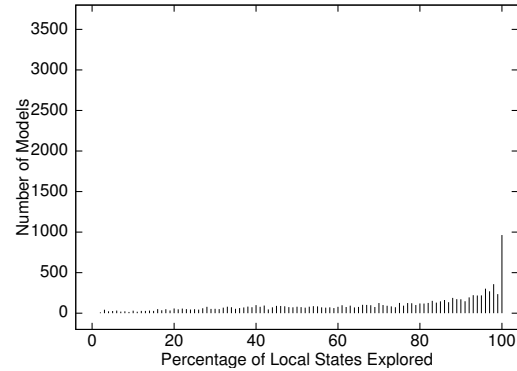
ber of inputs per transition exceeds 250 we see % explored values only above 60%. So TAR2’s suggested treatment, that we restrict the number of inputs per transition to the highest range, makes sense.

The real power of the TAR2 treatment learning approach is in more complex treatments, which suggest restrictions on multiple attributes. Figure 12 shows a comparison of % explored (our indicator of Lurch’s ability to successfully search a particular model) for the original data (top) and a new set of 10,000 input models (bottom) generated with input parameters based on TAR2’s best three-attribute treatment. Figure 12 shows what we expect: a clear improvement in Lurch’s ability to search models constrained to the suggested treatment.

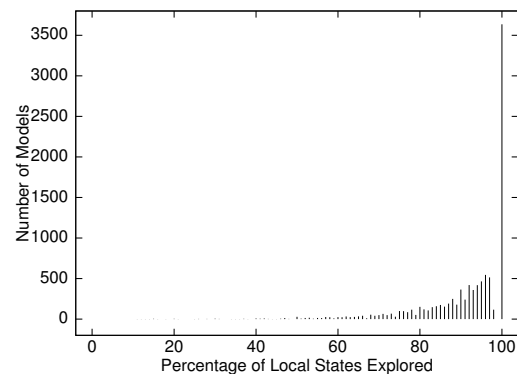
## 11 Emergent Behavior and Design

Elsewhere the experiments and ideas presented above have been used to argue certain points in software engineering [14, 17, 18]. Here we will try to offer a broader application: what does the the success of random search in exploring software models tell us about the relationship between emergent behavior and intelligent design?

Often the analogy is made between naturalistic evolution and algorithms used in various computer applications. Just as we are able to solve very complex problems by repeating a relatively simple set of steps, the continual process of evolution—reproduction with variation, natural selection—is apparently able to produce amazingly complex new forms of life. And just as many computer algorithms work automatically to solve problems, without intervention from the user or programmer, evolution seems to work without inter-



Original search data (scale adjusted to match new search data plot immediately below)—average % explored = 69.4%.



Search data for input models generated according to TAR2’s suggested treatment—average % explored = 91.34%.

**Figure 12. Comparison of plateau height for original search data (top) and new data based on TAR2’s suggested treatment.**

vention from any intelligent mind or designer. For example, in *Darwin’s Dangerous Idea* Dennett uses this kind of analogy to summarize his view of the interaction between proponents and skeptics of gradual, naturalistic evolution:

Darwin’s dangerous idea is that Design can emerge from mere Order via an algorithmic process that makes no use of pre-existing Mind. Skeptics have hoped to show that a least somewhere in this process a helping hand (more accurately, a helping Mind) must have been provided—a skyhook to do some of the lifting [here “skyhook” refers to a mythological device for supernaturally lifting objects from the sky—Dennett’s metaphor for miraculous intervention from an intelligent mind or designer]. In their attempts to provide skyhooks, they have often discovered

cranes: products of earlier algorithmic processes that can amplify the power of the basic Darwinian algorithm, making the process locally swifter and more efficient in a nonmiraculous way [6].

Recall the ants-and-trees metaphor in section 8. Individual ants are like the individual finite-state machines in a software model. The tree represents the space of all possible behaviors of the interacting ants. A random search verification tool like Lurch explores the emergent behavior of the individual parts of a software model as they interact and influence each other through time—in a sense, tracking the ants as they move through the tree. In a software model the individual parts operate according to a pre-defined set of legal transitions. It is not possible to simply jump from one state to another, just as it is not possible for an ant to spontaneously disappear from one part of the tree and reappear in some remote part. This sort of miraculous behavior would require one of Dennett’s “skyhooks.”

And yet the success of Lurch on a software model—the ability of the ants to reach some interesting part of the tree without spontaneous leaps—this in no way precludes intelligent design of the overall system. In fact, as we have seen above, it is possible to design a system in such a way that the right sort of behavior emerges. Given a range of possible software models we can sort them according to their tendency to reveal information and determine which sort of models are amenable to random search. To put it another way, the emergent behavior of the ants is greatly influenced by what sort of tree they are exploring (figure 7).

When we observe that a relatively simple process is surprisingly effective, whether in software engineering, biology or any other field, we should not accept the result uncritically as proof that there is no designed structure present in the system. Perhaps the tree has been designed (or is continually modified) to influence the emergent behavior of the climbing ants; perhaps it is possible to design software that tends to reveal its behavior to random search algorithms—and it could be that the success of evolution is evidence for, not against, the existence of an intelligent designer. When presented with interesting emergent behaviors, we should always ask: what does a particular behavior tell us about the structure of the space of possible behaviors—what can we learn from a shot in the dark?

## 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*, 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, 1999.

[5] S. Curtis, J. Mica, J. Nuth, G. Marr, M. Rilee, and M. Bhat. ANTS (Autonomous Nano-Technology Swarm): An Artificial Intelligence Approach to Asteroid Belt Resource Exploration. In *International Astronautical Federation, 51st Congress*, 2000.

[6] D. Dennett. *Darwin’s Dangerous Idea: Evolution and the Meanings of Life*. Simon and Schuster, 1995.

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

[10] T. Menzies and B. Cukic. Intelligent Testing can be Very Lazy. In *The First International Workshop on Intelligent Software Engineering*, 1999.

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

[12] T. Menzies and B. Cukic. When to Test Less. *IEEE Software*, 2000.

[13] T. Menzies and Y. Hu. Constraining Discussions in Requirements Engineering via Models. In *First International Workshop on Model-Based Requirements Engineering*, 2001.

[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 *Second International Workshop on Soft Computing Applied to Software Engineering*, 2001.

[16] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[17] 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.

[18] D. Owen and T. Menzies. Lurch: a Lightweight Alternative to Model Checking. In *The International Conference on Software Engineering and Knowledge Engineering*, 2003.

[19] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.