God Does Play Dice: Diagnosis and Validation for Autonomous Systems

S. Bayana[1] David Owen[2] Tim Menzies[2] Supratik Mukhopadhyay[2] [*†‡]

Abstract

Randomized algorithms have been known to outperform their deterministic counterpart over a wide range of problems. In this paper, we use randomized techniques for validating and diagnosing autonomous intelligent systems. Such techniques provide efficient approximate solutions to both the diagnosability and the validation problems. In particular, we show the effectiveness of LURCH, a randomized inference engine that we have developed in validating and diagnosing autonomous systems. LURCH uses random search methods that use (1) a fast partial search, (2) a random selection amongst options, and (3) the occasional reset/restart. We have conducted case studies on an optical navigation system, a camera control system and several components a propulsion system all written in a Reactive Model Programming Language (RMPL).

1. Introduction

Intelligent software is now being considered more often as a vehicle for providing greater autonomy to automated systems, replacing humans, in places where they cannot or would not venture themselves, with robots. Such a trend is best exemplified in NASA's missions that continue to explore Mars and beyond. The great distances from earth will require that they will be able to perform many of their tasks autonomously. Autonomous systems rely on intelligent inference capabilities to be able to take the right actions even in unknown environments. They perform many of their tasks autonomously; e.g., the autonomous controller for the in-situ propellant production facility, supposed to produce spacecraft fuel on Mars, must operate with infrequent, severely limited human intervention to control complex, real time, and mission-critical processes over many months in poorly understood environments. While autonomy offers promises of improved capabilities at a reduced operational cost, development and validation of software for such autonomous systems poses a tough challenge due to exponential blow up in the number of possible situations that they need to deal with.

One of the most effective techniques for dealing with the complexity of developing software for autonomous systems is *model-based programming*. It is based on the observation that programmers and operators generate the breadth of desired functionality from common-sense hardware models in light of mission level goals. Development of model libraries reduces design time, facilitates reuse and amortizes software development costs. More importantly, validation can be done at an early stage of the software design cycle. Several high level languages for model-based programming have been developed such as JMPL [1], RMPL [28] etc.

Autonomous systems have been recently integrated into robotic networks. Model-based methods [11, 25, 27–29] have been developed to monitor and coordinate such complex autonomous systems and automate their *diagnosis*. Diagnosis is an important component of autonomy for any intelligent agent. Often, an intelligent agent plans a set of actions to achieve certain goals and because some conditions may be unforseen, it is important for it to be able to reconfigure its plan depending upon the state in which it is. This state identification problem is essentially a problem in diagnosis. In a model-based diagnosis and monitoring methodology, a model of the correctly functioning system is used to predict the system's behavior. Discrepancies between these predictions and the observations of the system are symptoms; the rest of the diagnostic and monitoring task (hypothesis formation, refinement and testing) is driven by these symptoms (as in a feedback control system). Automated model-based diagnosis has the capability to diagnose and explain *novel faults*. System (or system component) modes that represent failures are assigned a cost corresponding to the prior probability of that failure occurring in that system (or system component). Starting with the lowest cost mode assignment, all complete mode assignments in order of total likelihood are considered until an assignment cosistent with the observations is found. This mode assignment represents the most likely state of the system.

Model-based diagnosis and recovery systems like Livingstone [29] have demonstrated their utility in several NASA missions. In particular, Livingstone formed a com-

[*1] West Virginia University
[†2] NASA/WVU Software Research Laboratory
[‡]This research is supported by NASA under the OSMA SARP Program

ponent of the Remote Agent architecture deployed in the Deep Space probe. Livingstone monitors the sequence of discrete commands that are issued to the system to track the expected state of the system and compare the predictions generated from its model against the observations received from the sensors. Once a discrepancy occurs, Livingstone performs diagnosis by searching for the most likely set of (system) component mode assignments that are consistent with the observations.

As mentioned earlier, software for autonomous systems pose a tremendous challenge in terms of validation and diagnosability due to combinatorial explosion in the number of possible situations. Traditional deterministic search strategies are unlikely to be efficient due to the inherent exponential blowup. In te realm of validation, traditional testing methods fail to provide the desired confidence level due to the combinatorial explosion in the number of possible paths. Formal methods like model checking [4] and theorem proving [22] fail to scale up due to the exponential blowup in the size of the state (proof) spaces.

Randomized algorithms have been known to outperform their deterministic counterpart over a wide range of problems [19]. A randomized algorithm is allowed access to a source of independent, unbiased, random bits. It is then permitted to use these random bits to influence its computation. For many problems that require searching large spaces of solutions, randomized techniques are known to provide efficient solutions, while the best algorithms that their deterministic counterparts can come up with are those that run in exponential time. Popular graph theoretic problems that arise frequently in practice, such as the graph isomorphism problem and the perfect matching problem fall in this class.

In this paper, we use randomized techniques for validating and diagnosing autonomous intelligent systems. Such techniques provide efficient approximate solutions to both the diagnosability and the validation problems. In particular, we show the effectiveness of LURCH, a randomized inference engine that we have developed in validating and diagnosing autonomous systems. LURCH uses random search methods that use (1) a fast partial search, (2) a random selection amongst options, and (3) the occasional reset/restart. The cost of randomized inference is its inaccuracy. If complete inference terminates, it will find (infer) all the features that are deducible. On the other hand, by their very nature, randomized inference engines 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. Due to lack of space, a theoretical analysis of LURCH is beyond the scope of this paper. Some analytical results offering us some confidence in the generality of LURCH-style inference can be found in [18].

We have conducted case studies on an optical navigation system, a camera control system and several components a propulsion system all written in a Reactive Model Programming Language (RMPL) [28]. RMPL is a high-level model-based programming language that can express a rich set of hardware and software behaviors. The optical navigation system as well as the camera control system was a component of NASA's Deep Space One Probe (DS1). The propulsion system is part of NASA's Propulsion IVHM (Integrated Vehicle Health Management) Technology Experiment (PITEX). We present a translator that convert models written in RMPL into specifications that can be input to LURCH. It thereby shields the system designer from the technicalities of LURCH.

The rest of the paper is organized as follows. In Section 2 we provide a brief introduction to the RMPL language. Section 3 provides an introduction to LURCH and its implementation. It argues about the benefits of random search. Section 4.1 discusses the use of LURCH for model-based diagnosis and describes experimental results. Section 4.2 describes use of LURCH as a validation tool. Related work is discussed in Section 5. Finally, Section 6 concludes the paper.

2 RMPL

RMPL is a high level model programming language that merges constructs from synchronous programming languages, qualitative modeling, Markov models and constraint programming. It consists of a minimum set of primitives for constructing programs. A variety of program combinators are defined on the top of these primitives to make the language usable. The primitive constructs of RMPL are as follows (A, B represent well-formed RMPL programs).

- c (constraint or state) This construct asserts that the constraint or state holds at the initial instant of time.
- **if** c **thennext** A. This program starts behaving like A in the next instant if at the current instant the constraints holding entail c.
- **unless** c **thennext** A. This program executes A in the next instant if the constraints at the current instant do not entail c. This construct is used for preemption. It allows A to proceed as long as some condition is unknown but stops when the condition is determined.
- A, B. This program executes two concurrent processes A and B.
- **always** A. This program starts a new copy of A at each instant of time, for all time.
- **choose**[A **with** p, B **with** q]. This program reduces to A with probability p and to B with probability q where $p + q = 1$.

These six primitive combinators can be used to implement a rich set of combinators. RMPL provides full concurrency, conditional execution, iteration, premption, probabilistic choice and co-temporal constraint. An example of a RMPL program is given below.

```
Camera :: always {
choose {
{
if CameraOn then {
if TurnCameraOff thennext MICASoff
elsenext CameraOn ,
if CameraTakePicture thennext CameraDone
} ,
if CameraOff then
if TurnCameraOn thennext CameraOn
elsenext CameraOff ,
if CameraFail then
if MicasReset thennext CameraOff
elsenext CameraFail
} with 0.99 ,
next CameraFail with 0.01
}
}
```

It describes the camera control system shown in Figure 1. The dotted lines in the figure indicate threads starting simultaneously. The texts beside the circles indicate the constraints (boolean) true at the corresponding states. The texts on the arrows indicate the constraints that must hold for the transition to take place. From the initial state the system either evolves to the state $CameraFail$ with probability 0.01 or to a state with probability 0.99 where it evokes three threads indicated. The first thread in turn invokes two new threads at the next time instant. The program exploits full concurrency by intermingling sequential and parallel threads of execution. The full RMPL language is an object-oriented language in the style of Java that supports all the primitive combinators and a variety of defined combinators. For more details about RMPL consult [28].

3 Lurch

As alluded to before, LURCH is a randomized inference engine. We first introduced LURCH in [21]. While complete search or inference is prefered, some models are too large to be processed by complete search 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. LURCH-style inference was very simple to implement. For example, our current version of LURCH is less than 1000 lines long. It is written in C and uses hashing techniques for fast search.

LURCH takes as input statecharts modeled as transition functions.

Difficult search problems, e.g., *NP-hard* problems, have been shown to exhibit a *phase transition* (figure 2) [2, 6, 16]. 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" [2].

Figure 3 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 [21]. 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.

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 [8].

The complete inference technique may be overkill, however, for problems which turn out to be in the easily solvable range (recall figure 2). The algorithm is described in brief in figure 4 (for full details, see [17,20,21]). LURCH uses an memory-saving AND-OR graph representation of the composite system behavior[1]. LURCH's search space contains

[1]To justify the analogy between LURCH results and phase transition results reported by others, note that complete search of the AND-OR graph used by LURCH to represent the composite system is in fact NP-hard [20].

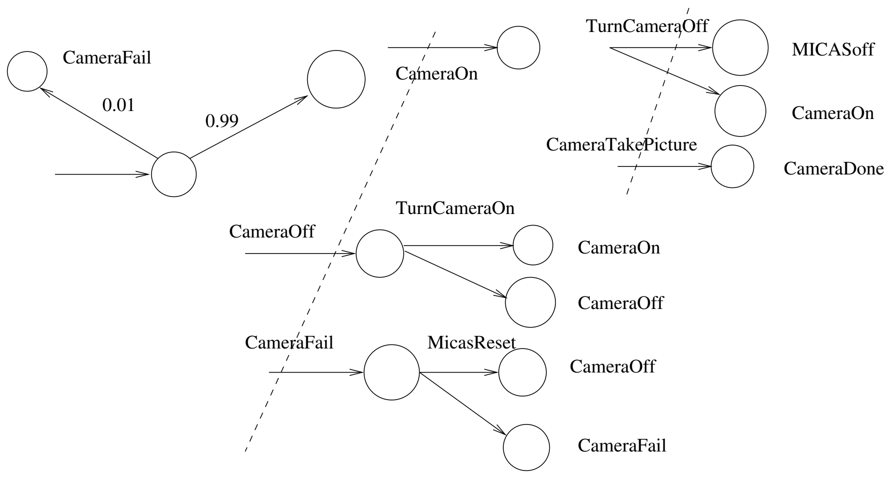

Figure 1. Camera Control System

$A * V$ number of nodes: i.e. one node for each possible assignment A to a every variable V. By contrast, the search space of a complete inference engine contains at most A^V nodes: i.e. one node each consistent set of assignments to all variables.

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 3, and, as indicated by the experiments presented below, LURCH is surprisingly successful compared to more sophisticated inference tools.

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 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 [17]) 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.[2]

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 2 and 3: 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.

[2]For very large input models, 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.

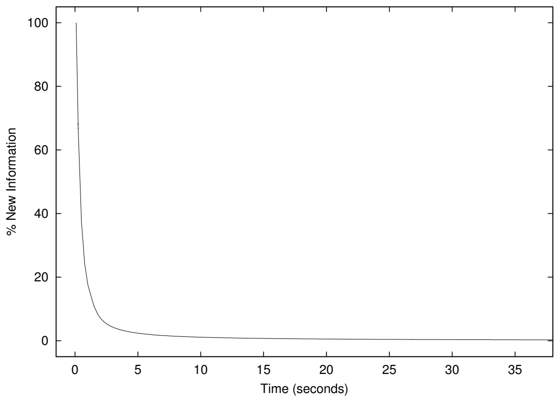

Figure 5. LURCH output for a typical model: quick saturation.

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.

In order to counter state explosion, finite state machines are translated to And-Or graphs during their representation in LURCH. The translation procedure assumes that finite-state machines have transitions defined as follows: each transition begins in the *current state*, and takes place if all *inputs* are true. If a transition takes place, all outputs are set true and the machine moves to the *next state*.

In an AND-OR graph, an AND-node is considered true if all of its parent nodes are true; an OR-node is considered true if any one of its parents is true. We add NO-edges to indicate which nodes may not be true at the same time. For example, since a local machine may not be in two states at once, NO-edges connect all OR-nodes representing states in the same local machine. Also, since two transitions in a local machine may not occur simultaneously, NO-edges connect all AND-nodes representing transitions in the same local machine. The resulting AND-OR graph has $O(n)$ nodes and $O(n^2)$ edges, where n is the size of the input [20].

Unfortunately, complete search of our AND-OR graphs is NP-hard and would therefore require exponential time

```
 1: node {
 2: kids, NO-kids;
 3: disqualified, frontier, reached;
 4: wait; }
 5: process-queue(time) {
 6: while (Q ≠ ∅) do
 7:   n ← pop(Q);
 8:   if (n.disqualified ≠ time) then
 9:     if (n is an OR-node) then
10:       n.reached = time;
11:       for (∀ nodes n′ ∈ n.NO-kids) do
12:         n′.disqualified = time;
13:     for (∀ nodes n′ ∈ n.kids) do
14:       n′.wait ← n′.wait − 1;
15:       if (n′.wait = 0) then
16:         if (n′ is an OR-node) then
17:           n′.frontier = time;
18:         else if (n′ is an AND-node) then
19:           Q ← n′ at random index; }
20: search() {
21: time = 0;
22: while (¬(path-end or cycle)) do
23:   process-queue(time);
24:   for (∀ nodes n : n.reached = time − 1) do
25:     if (n.disqualified ≠ time) then
26:       Q ← n at random index;
27:   process-queue(time);
28:   for (∀ nodes n : n.frontier = time) do
29:     Q ← n at random index;
30:   time ← time + 1; }
31: main() {
32: for (i = 1 to MAX-PATHS) do
33:   for (∀ nodes n) do
34:     n.disqualified ← n.frontier ← n.reached   ← UNDEF;
35:     reset n.wait to initial value;
36:   for (∀ nodes n : n is true initially) do
37:     Q ← n at random index;
38:   search(); }
```

Figure 6. Partial, random search procedure for AND-OR graphs.

[20]. Figure 6 shows the fast partial, random search procedure LURCH uses to search AND-OR graphs. As stated above, the search procedure is partial in that there is no guarantee that the entire AND-OR graph will be explored. Each iteration (of the *search* function, beginning on line 20 in figure 6) finds one global state path. With many iterations it becomes likely that nearly all of the reachable state space is explored. The procedure is random in that, when two or more paths may be explored, the choice is made based on the order nodes are popped from the queue (line 7). Since nodes are always pushed at a random index (lines 19, 24, 29, 34), the choice of which path to explore is nondeterministic.

For each node (Figure 6, line 1), *kids* and *NO-kids* (line 2) are lists of children via normal and NO-edges. The *disqualified*, *frontier*, and *reached* fields (line 3) mark at what time during the search a node is disqualified, part of the frontier, or reached; *wait* (line 4) is an integer indicating how may parents still need to be reached before the node is reached—*wait* is initialized to 1 for an OR-node and, for an AND-node, to the number of parents it has.

4 Diagnosis and Validation of Autonomous Ststems

Given a model of a physical system and a sequence of commands and observations received over time, a conventional diagnosis system determines the belief state (i.e., likely states of the system) and the actions required to move the system to a desired configuration. Computing a belief state (i.e., a probability distribution over the possible states of the system) entails enumeration of the state space. Such an approach is likely to suffer from the state explosion problem but for the simplest models. Diagnosis systems like Livingstone [29] focus on monitoring and diagnosing networks whose components have simple behaviors.

Validation tools like SMV, SPIN [7,14] have been used to detect when an undesirable state is reached. While they can be used [3] to detect violation of diagnosability by exhibiting a pair of paths that are indistinguishable but hide conditions that should be distinguished, such tools cannot be used for the actual state estimation problem that takes into account a probability distribution over the possible states of a system. Like their diagnosis counterparts, validation tools like SMV, SPIN etc. suffer from the combinatorial explosion in the state space.

LURCH combines the best of both worlds. On one hand LURCH can test a system detecting deviations from the behavior predicted from the model. This functionality can be used for diagnosis. On the other hand, LURCH can function as a partial model checker performing an approximate validation of the model. Due to its partial search strategy, it is less likely to suffer from the combinatorial explosion problem compared to the complete search techniques mentioned above.

4.1 LURCH for Diagnosis

In order to use LURCH to diagnose autonomous systems, we need to convert the models to the input format of LURCH. To this end, we have built a translator from RMPL to LURCH. The translator is written in Awk and consists of 1100 lines of code. The output of the translator for the camera control system shown in Section 2 is shown below.

```
CM5;     -;  p*=0.99;   CM5;
CM5;    (Camera_Signal==C_ON &&
TurnCamera==TurnC_OFF);
{MICAS=MI_OFF;};CM5;
CM5;    (Camera_Signal==C_ON &&
TurnCamera==TurnC_ON);
{Camera_Signal=C_ON;};  CM5;
CM5;    (Camera_Signal==C_ON &&
CameraTakePicture==TRUE);
{Camera_Signal=C_DONE;};    CM5;
CM5;    (Camera_Signal==C_OFF &&
TurnCamera==TurnC_ON);
{Camera_Signal=C_ON;};  CM5;
CM5;    (Camera_Signal==C_OFF &&
TurnCamera==TurnC_OFF);
{Camera_Signal=C_OFF;fault+=1;};CM5;
CM5;    (Camera_Signal==C_FAIL &&
MICAS==MI_RESET);
{Camera_Signal=C_OFF;}; CM5;
CM5;    (Camera_Signal==C_FAIL &&
MICAS!=MI_RESET);
{Camera_Signal=C_FAIL;;fault+=1;};CM5;
CM5;            -;
{Camera_Signal=C_FAIL;
p*=0.01;fault+=1;};CM5;
```

A transition in the LURCH input language is of the form *Source-State;Pre-conditions;Sideeffects;Target-State*. A precondition is a conjunction of constraints (possibly empty indicated by $-$) while a side effect is a sequence of C statements (possibly empty). Notice that the probability of a path is represented by a variable p which is updated (in the side effect) as a transition is taken (initially, $p = 1$). This assumes that the underlying state machine is a Markov machine. The number of faults occurring in a path is represented by the variable $fault$. In the above example, the mode C_FAIL of $Camera\_Signal$ is a faulty mode. Whenever a faulty mode is reached, the variable $fault$ is incremented. Thus, at the end of a path, the variable p will hold the probability of taking that path while the variable $fault$ will contain the number of faulty modes encountered in the path. In order to track the history of each path, we associate with it a cost $c = p/(fault + 1)$ which is a measure of how likely the path is and how desirable it is. The diagnosis problem is then to determine the path with the maximum cost.

Rather than computing a complete belief state (i.e., the likelihood of each possible configuration of the system), LURCH randomly chooses a trajectory (path) and computes the value of the cost c for that path. At the end of the trajectory, LURCH stores the trajectory as well as its cost in a buffer. This path is the current best path. If the cost of the next path traversed by LURCH is more than that of the current best path, then it replaces the latter. Trajectory extensions that are inconsistent with the current observation are ruled out. Thus LURCH tests a path for its value and updates the value of the current best path if the value of the currently tested path is more than that of the current best path. Since LURCH is a Monte Carlo algorithm, the test runs again and again, each tikme increasing the chance of finding the best path.

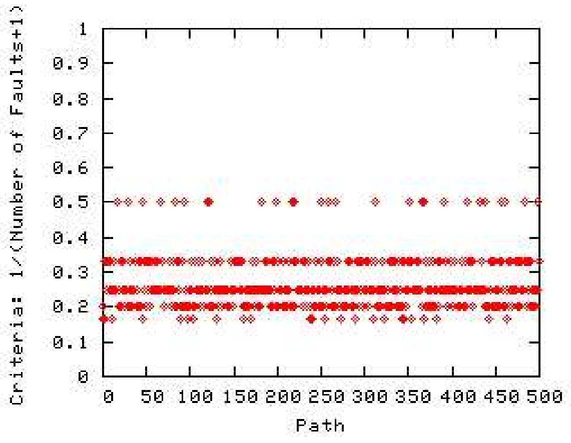

Figure 8. cost vs path number plot for propulsion system

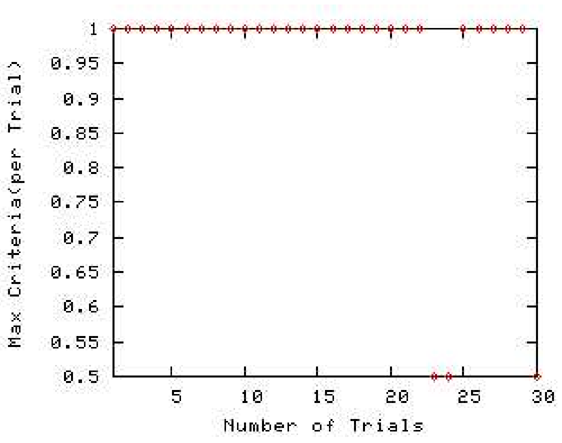

Figure 9. cost vs trial number plot for propulsion system

4.1.1 Diagnosis: Case Studies

We have conducted case studies in diagnosis on (1) the camera control system described in Section 2 (2) an optical navigation system [28] and (3) a portion (reactor and tach) of a propulsion subsystem for a spacecraft. The models for all the three examples were written in RMPL. For the propulsion subsystem, each component has on the average 3 states and has several faulty modes. The results of the experiments are summarized in Figure 7. All experiments were carried out on a 1.2 GHz Pentium PC running Linux.

In order to study how fast LURCH can converge to the best path, we plotted the cost vs path number graph for the propulsion system in Figure 8. It can be seen that the best path is found within the first 10 trajectories traversed. Since LURCH is a Monte Carlo algorithm, we also studied the output of LURCH for different runs (i.e., different values of the seed). We plotted the values of the cost obtained against the trial number for the propulsion system in Figure 9. A horizon of 30 trials was considered. The plot shows that (1) repeated trials with LURCH are needed to get the best path and (2) the best path is hit within the first few trials. We also studied the convergence behavior (i.e., the number of paths needed to converge to the best cost) of LURCH for different runs (i.e., different values of the seed) for the propulsion subsystem and found that LURCH uniformly converges to the best path (for a particular run) after exploring only a few paths.

4.2 LURCH for Validation

The random search of LURCH can be used as a partial model checker for validating the models. Unlike model checking, the partial search of LURCH is a quick analysis method. It does not suffer from the state explosion problem. We used LURCH's random search to determine whether fault states are reachable in the examples described above. For probabilistic models, we experimented with both blocking and non-blocking semantics. With the blocking semantics, for a probabilistic choice with probabilities p and $1-p$, a pseudorandom number is drawn in the interval $[0, 1]$ whose value determines which choice is enabled. If the number lies in the interval $[0, p]$, the first choice is enabled while the other is blocked while for the value of the number in the interval $[p, 1-p]$ the converse holds true. For the non-blocking semantics, transitions are taken independently. The probability of reaching a state is the product of the probabilities of all transitions on a path to the state from the initial state. LURCH was run several hundred times on each example. In each iteration of the search, we use a random set of consistent nodes (i.e., a partial description of a global state) as input. Our output is a set of states from all iterations reachable from the input state. For all the three examples LURCH was able to detect reachability of faulty states quickly. In spite of the fact that LURCH implements a Monte Carlo algorithm, we were able to detect reachability of most of the faulty states in the first run itself. Only a fraction of the state space of the models were searched. In case of the propulsion subsystem, LURCH was able to reach two fault states in the first run itself exploring a total of 43 global states and 32 transitions.

5 Related Work

Recently, there has been a surge of activity in model-based diagnosis and validation of autonomous systems. Such activity has been spurred by NASA's missions which continue sending robotic explorers to space. Model-based diagnosis traces its root back to Reiter's seminal paper [25]. Since then several generic systems for model-based diagnosis have been developed using logical inference, assumption-based truth maintenance, and conflicts as their underlying principles (see [5]). The works in model-based diagnosis that come closest in spirit to ours are [9–12, 24, 27–29]. Williams and Nayak [29] describe Livingstone, an implemented kernel for a model-based reactive self-configuring autonomous system. As mentioned earlier, Livingstone has been successfully deployed in several NASA missions including the Deep Space probe. Livingstone works by enumerating only the most likely portion of the belief state at each point in time by transitioning a small number of tracked states by the transitions that are most likely, given the current observations. Such an approximation is extremely efficient and well suited to the problem of tracking the internal state of the machine, where the likelihood of the expected transition dominates, and immediate observations often rule out the expected trajectory when a failure occurs. But as shown in [11], the true trajectory (path) may not be among the most likely trajectory given only the current observations. Kurien and Nayak [10, 11] propose to maintain the information necessary to begin incrementally generating the current belief state in best first order at any point in time. Since they do not update the entire belief state, a history must be maintained. In contrast with [10, 11, 29], LURCH is a randomized inference engine that "tests" a path for a cost. Besides, Livingstone, like other diagnostic systems focuses on monitoring networks whose components have simple behaviors. Trajectories that spend their time wending their way through a mixture of software and hardware functions are well beyond the capabilities of Livingstone. LURCH transitions have the capability of calling functions written in ordinary C code; this facility gives LURCH the capability to deal with such trajectories involving a mixture of hardware and software functions.

In his PhD thesis [27], Throop extends model-based diagnosis to continuous systems. Applying LURCH to continuous systems is a topic of our future. Williams et. al [28] introduce RMPL a rich modeling language that combines reactive programming constructs with probabilistic constraint-based modeling. To support mode diagnosis, they translate RMPL models to hierarchical constraint hidden Markov models (HMMs). They extend traditional HMM belief update to track a system's most likely states. While the search procedure of [28] is a complete search, LURCH is based on an incomplete partial search. Kumar [9] provides a unifying theme behind all the approaches to model-based diagnosis based on the notion of model counting. It is intresting to see how LURCH fits this theme. Lucas [12] develops methods for reasoning with uncertainty in consistency-based dignosis by integrating logical reasoning as done in consistency-based diagnosis and probabilistic reasoning as done in Bayesian networks. Poole [24] introduces partial evaluation techniques in probabilistic inference. It is interesting to see if such techniques can be incorporated in a randomized inference engine like LURCH. Menzies et. al [15] proposed testing a theory (instead of a complete search) to check if it can produce its known behavior. This paper can be thought of using the same principle for model-based diagnosis.

In terms of validation of models, the works that come closest to us are [3, 13, 23, 26]. In [3], Cimatti et. al address the problem of diagnosability: given a partially observable dynamic system, and a diagnosis system observing its evolution over time, [3] addresses the problem of verifying if the diagnostic system will be able to infer the required information on the hidden part of the dynamic state. They recast this problem in the framework of model checking and use symbolic model checking to solve the problem. Pecheur et. al. [23, 26] use SMV to validate Livingstone models. They automatically translate Livingstone models written in a model programming language to the SMV input language. While SMV performs a complete search of the state space of the model, being, thereby, vulnerable to the state explosion problem, LURCH, as discussed above avoids the problem of exponential blowup in the search space by conducting a partial search. Besides, [23, 26] deal with nondeterministic models while we deal with probabilistic models as well. Kwaitkowska et. al [13] describe PRISM a probabilistic extension of SMV. It would be interesting to see how PRISM would perform on models of autonomous systems. In [21], we introduce LURCH as an alternative to model checking.

6 Conclusion

In this paper, we described the application of randomized techniques for disgnosing and validating autonomous systems. We applied a randomized inference engine LURCH for diagnosing and validating autonomous systems with models written in RMPL. Unlike conventional model-based diagnosis that computes a belief state by a complete search of the state space, LURCH performs a partial random search. Preliminary experiments using LURCH as a tool for validating and diagnosing autonomous systems have shown encouraging results. We intend to perform additional case studies to confirm the effectiveness of LURCH as a model-based diagnosis system and plan to compare the performance of LURCH with complete search procedures. Future work also includes analysis of correlation between the search saturaation and the design structures that lead to it. Finally, we plan to explore techniques from stochastic game theory to diagnose and validate partially specified systems.

References

[1] A. Bachmann. Introduction to jmpl.

[2] P. Cheeseman, B. Kanesfy, and W. Taylor. Where the *Really* Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence IJCAI–91, Sidney, Austrailia*, 1991.

[3] A. Cimatti, C. Pecheur, and R. Cavada. Formal verification of diagnosability via symbolic model checking. In *Proceedings of the International Joint Conference on Artificial Itelligence: IJCAI'03*, 2003.

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

[5] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:157–208, 1994.

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

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

[8] 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`.

[9] T. K. S. Kumar. A model counting characterization of diagnoses. In *Proceedings of the Thirteenth International Workshop on Principles of Diagnosis (DX'02).*, 2002.

[10] J. Kurien. *Model-based Monitoring, Diagnosis and Control*. PhD thesis, Brown University, 2000.

[11] J. Kurien and P. Nayak. Back to the future for consistency-based trajectory tracking. In *Proceedings of the National Conference on Artificial Intelligence: AAAI'00*. AAAI Press, 2000.

[12] P. Lucas. Bayesian model-based diagnosis. *International Journal of Approximate Reasoning*, 27(2):99–119, 2001.

[13] G. N. M. Kwiatkowska and D. Parker. Prism: Probabilistic symbolic model checker. In *Proc. TOOLS 2002*, LNCS. Springer, 2002.

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

[15] T. Menzies, R. F. Cohen, S. Waugh, and S. Goss. Applications of abduction: Testing very long qualitative simulations. *IEEE Transactions on Knowledge and Data Engineering*, 14(6):1362–1375, 2002.

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

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

[18] 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`.

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

[20] D. Owen. Random Search of AND-OR Graphs Representing Finite–State Models. Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University, 2002.

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

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[23] C. Pecheur, R. Simmons, and P. Engrand. Formal verification of autonomy models: From livingstone to smv. In *Proceedings of First Goddard Workshop on Formal Approaches to Agent-Based Systems, NASA Goddard*, LNCS. Springer, 2000.

[24] D. Poole. Probabilistic partial evaluation: Exploiting rule structure in probabilistic inference. In *PProceedings of the International Joint Conference on Artificial Itelligence: IJCAI'97*, 1997.

[25] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[26] R. Simmons, C. Pecheur, and G. Srinivasan. Towards automatic verification of autonomous systems. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE*, 2000.

[27] D. R. Thropp. *Model-based Diagnosis of Complex Continuous Mechanisms*. PhD thesis, University of Texas at Austin, 1991.

[28] B. C. Williams, S. Cheung, and V. Gupta. Mode estimation of model-based programs: Monitoring systems with complex behavior. In *Proceedings of the International Joint Conference on Artificial Intelligence: IJCAI 2001*. AAAI Press, 2001.

[29] B. C. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the National Conference on Artificial Intelligence: AAAI'96*. AAAI Press, 1996.

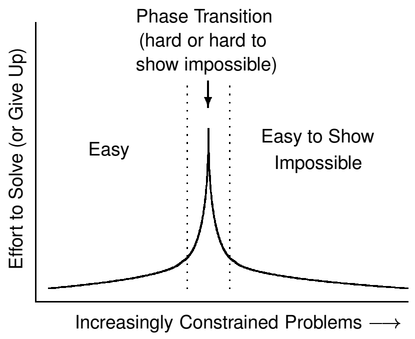

Figure 2. Hard problems exhibit a phase transition.

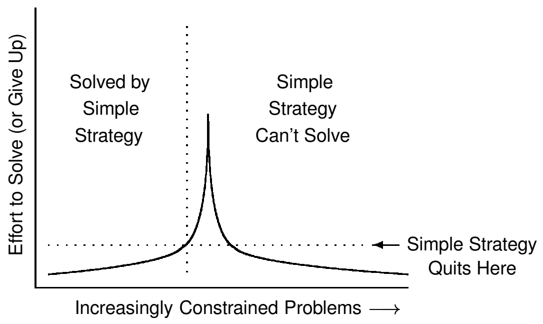

Figure 3. The power of a simple solution strategy.

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

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

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

Figure 4. LURCH's partial, random search procedure.

Example	Time	Cost	Maximum Depth	Number of paths	Local States	Transitions
Camera	0.00	0.004950	16	500	18	17
Propulsion	0.01	0.5	21	500	47	36
Navigation	0.00	0.33	2	500	14	10

Figure 7. Experimental Results