Testing Nondeterminate Systems

Tim Menzies
NASA/WVU Software Research Lab,
Fairmont, USA
tim@menzies.com

Bojan Cukic
Dept. Com. Sci. & Elec. Eng.
West Virginia University,
Morgantown, USA
cukic@csee.wvu.edu

Harhsinder Singh
Dept. of Statistics,
West Virginia University
Morgantown, USA
hsingh@stat.wvu.edu

John Powell
Dept. Com. Sci. & Elec. Eng.
West Virginia University,
jpowell@csee.wvu.edu

Abstract

The behavior of nondeterminate systems can be hard to predict, since similar inputs at different times can generate different outputs. In other words, the behavior seen during testing process may not be seen at runtime.

Due to the uncertainties associated with nondeterminism, the standard view is that we should avoid such nondeterminate systems, especially for systems requiring high reliability. While this is a valid guideline, at least in two application areas such nondeterminacy is unavoidable. Early life cycle requirements and AI software are becoming widely used. Yet both are imprecise and may exhibit nondeterminate behaviour if explored rigorously by a test device.

Based on a literature review and some theoretical studies, we argue that many stable properties exist within the space of all possible nondeterminate behaviors. However, we also show that seemingly trivial changes to a nondeterministic system can turn an easily testable system into an impossibly hard system to test. Finally, we stress that this analysis does not imply a correlation between stable zones of nondeterminate testability and the ultimate maintainability of nondeterminate systems. That is, while we are optimistic about testing nondeterminate systems we remain cautious about the maintenance of such systems.

1. Introduction

Despite the problems with determining their reliability, nondeterministic systems appear frequently in modern software applications. By “nondeterministic systems”, we mean those systems which, when presented with the same inputs at different times, may generate different output. That is, the output of such systems is not uniquely determined by the inputs.

It can be very difficult to quantitatively assess software reliability from such systems. In a sampling model, for example, software reliability is estimated as the probability of drawing a black ball (signifying a point in the input space which reveals a program failure) from an urn containing black balls and white balls (points in the input space which reveal no failures) [16]. However, this view is incorrect if the same input, due to nondeterministic program execution, may result in different outputs (correct or incorrect). The correct analogy for nondeterministic systems is to view each ball as consisting of several fragments (say N), each of which can be either black or white. Further, each time a ball is selected, only one fragment can be examined. Under these conditions, estimating the probability of failure based on this type of testing does not follow well established mathematical models. The same reasoning holds for all quality models which treat the system under test as a black box. The lack of determinism in the input-output behavior of the program introduces uncertainty. As a consequence, users, if given a choice, prefer to avoid such implementations.

Nevertheless, we have observed an increasing use of nondeterministic systems, particularly in the fields of requirements engineering and artificial intelligence applications. Such increases are often mandated by economic considerations. For example, a repeated observation is that the removal of defects from requirements documents is orders of magnitude cheaper than removing defects from delivered source lines of code [19]. Because of the financial benefits of early testing, developers may thus be mandated to test their requirements as early as possible. Unfortunately, early life-cycle requirements tend to be under-specified, particularly if they come from multiple stake holders [9]. Hence, they are nondeterminate since the results they specify may be contradictory. At the NASA Independent Verification and Validation facility, we witness these type of requirements frequently, even though the final products of these specifications are safety and mission critical systems [19].

In AI, efficiency concerns also drive us to nondeterministic systems. Increasingly, AI applications use randomized inference. For example, scheduling algorithms derive schedules that can be built from some initial random guesses [7]. Surprisingly, larger problems can be solved with such random inference procedures than with a more thorough search [7, 20]. Hence, when processing large AI systems, developers may want to use randomized inference. If the scheduling problem allows for different solutions, the randomized search picks one of these solutions. Underconstrained scheduling problems increase the probability that the random search reveals one of these solutions, making nondeterminacy desirable. However, before relying on randomized search engines, developers need to assess their behavior and the implications within the context of the specific application.

The goal of this paper is to determine how long we should test nondeterministic systems. We will show that testing nondeterminate systems is not necessarily more difficult than testing deterministic ones. More precisely, we will suggest that the number of tests required to be (e.g.) 99% sure of exercising all parts of a nondeterministic system is not determined just by the presence or absence of nondeterminism. Certainly, nondeterminism is one factor in determining test set size. However, this factor is far less critical than others, such as the proportion of “and” nodes in the program representation, or the average number of paths to some part of a program. We will demonstrate this as follows:

- We begin by defining an abstract model of program execution in a nondeterministic system; i.e. *traversing a NAYO graph* (defined below).

- We then derive an expression for the odds of reaching some part of that system from sets of random inputs. This expression lets us compute N; i.e. the number of randomly selected inputs needed to have a high chance of *reaching* that part of a nondeterministic system.

- Next, by executing the expression for a wide range of systems, we can determine when that part of a nondeterministic system is very *reachable* or not very reachable at all. We will show that for a large class of systems, most parts of a nondeterministic system can be exercised by a small number of random inputs. That is, for *easily reachable nondeterministic systems*, we can quickly sample all of their behaviour.

```
diet(fatty).
diet(light).
happy            :- tranquillity( hi).
happy            :- rich , healthy.
healthy          :- diet(light).
satiated         :- diet(fatty).
tranquillity(hi) :- satiated.
tranquillity(hi) :- conscience(clear).
```

Figure 1. Some ground horn clauses.

As a result of this work, we are optimistic about our ability to quickly test nondeterministic AI systems and requirement models. Our work shows that there exist large classes of nondeterministic systems for which we can quickly sample their space of behaviours using random inputs.

However, this work also makes us cautious about maintaining nondeterministic systems. We will show that that minor changes to program structure can have major changes to reachability. For example, instead of requiring (e.g.) less than 100 random tests, developers may suddenly need 1,000,000 tests or more to explore all of their system.

1.1. Preamble

Before beginning, we will describe some boundaries on this analysis.

Firstly, we only comment on those systems that can be expressed in our *NAYO graphs*. This includes two commonly used types of systems:

- We will show below that the *horn clauses* used in logic programming, expert systems, and much of AI can be expressed in our format. Horn clauses are also often used in logic-based approaches to requirements engineering. For an example of horn-clauses, see Figure 1.

- Finite-state diagrams can be reduced to horn clauses (see Figure 2) and hence can be expressed in our format. Finite-state diagrams can be found in many analysis methods or can be automatically derived from a static analysis of a program.

Secondly, in this paper, testing is viewed as the construction of pathways that *reach* from inputs to some interesting zone of a program. This zone could be a bug or a desired feature. In this *reachability* view, the goal of testing is to show that a test set uncovers no bugs while reaching all desired features. This reachability view is consistent with at least two testing regimes seen in the contemporary testing literature:

- Model checkers such as SPIN [10] generate trace files showing exactly how system constraints can be violated. Such traces clearly indicate how a program can

Horn clauses take the form

$$Goal\ if\ SubGoa11\ and\ SubGoal2\ and\ ...$$

which, in a Prolog notation, we would write as

$$Goal\ :-\ SubGoa11,\ SubGoal2,\ ...$$

If there exists more than one method of demonstrating some $Goal$, then each method is a separate clause.
Finite-state diagrams (FSDs) contain transitions between states. Transitions may be conditional on some guard. States may contain nested states.
To translate FSDs to horn-clauses, create one variable for each state. Create one clause for each transition from state $S1$ to $S2$. Each clause will take the form

$$S2\ :-\ S1,\ gaurd$$

where $guard$ comes from the conditional tests that activate that transition. If a state $S1$ contains sub-states $S1.1$, $S1.2$,... then create clauses of the form

$$S1.1\ :-\ S1$$

$$S1.2\ :-\ S1$$

, etc.

Figure 2. Translating finite-state diagrams to horn clauses.

fail. Also, the trace file can be used to explore methods of fixing the fault. In our framework, such a trace would be a tree extracted from the horn clauses whose root represented some error.

- One assessment of the utility of a test suite is how well it *covers* a program. Various definitions of coverage exist and one of the strictest is *DU-coverage*; i.e. covering all pathways from where a variable is set to where it is used [4]. In our example, a DU-path over horn clauses would be a tree extracted from the horn clauses whose root contains the usage. All the non-root members of that tree would contain variables that must be defined in order to reach that root. That is, reachability theory can compute the odds of generating a DU-pathway.

A third boundary is that thus far our analysis has not let us assess the testability of a *particular* nondeterminate system. Our reachability model refers to many parameters that describe a program and, at the time of this writing, we lack the tool set to extract those parameters from programs. The creation of that tool set is the current goal of this project. However, while we await that creation, we can still discuss classes of systems, even if we cannot discuss particular systems.

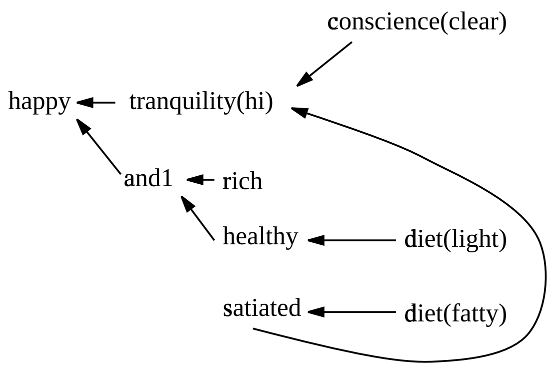

Figure 3. The and-or graph within Figure 1.

2. Traversing a NAYO Graph

2.1. Introducing NAYO Graphs

Our theoretical analysis of testing nondeterminate systems assumes that program execution and testing is a process of exploring a NAYO graph. This section describes the characteristics of NAYO graphs, while the next section will describe exploring NAYO graphs.

A NAYO graph is a finite directed graph containing two types of edges and two types of nodes:

- *Or-nodes* store assignments of a single value to a variable. Only one of the parents of an or-node need be reached before we visit the or-node.
- *And-nodes* model multiple pre-conditions. All the parents of an and-node must be reached before this node is visited.
- *No-edges* represent illegal pairs of inferences; i.e. things we can't believe at the same time. For example, we would connect `happy` and `sad` with a no-edge.
- *Yes-edges* represent legal inferences between or-nodes and and-nodes.

We can construct NAYO graphs from commonly-used representations such as the horn-clauses shown in Figure 1. Recall from the above that horn clauses form a special kind of system where each clause has a goal and sub-goals. Thus:

- To prove the clause's goal, we must recursively prove the items in the body. In Figure 1, we can prove `happy` in one or two ways. One method is to prove `rich` and `healthy`. Alternatively, we can prove `happy` if we can prove `tranquillity(hi)`.
- A clause with an empty set of sub-goals is a fact; i.e. we can believe it without further proof. In Figure 1, `diet(light), and diet(fatty)` are facts.

To convert this example to a NAYO graph, we first add one or-node for every term in Figure 1 plus one and-node for every non-empty body. We next add one edge for every body term connecting back to the head term. This procedure yields Figure 3.

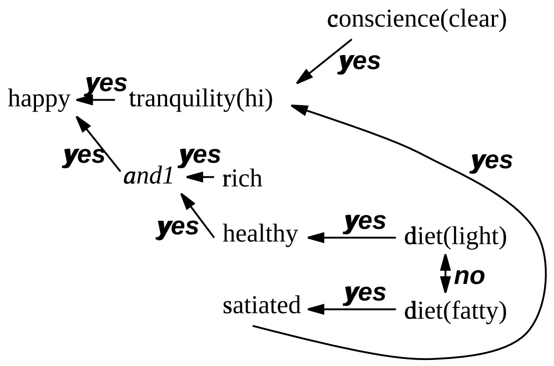

Figure 4. Figure 3 rewritten as a NAYO graph. All the nodes in this graph, except for `and1` are or-nodes.

To complete the build of the NAYO graph, we add the yes-edges and the no-edges. First, we label each edge in Figure 3 with "yes". Next, for each incompatible pair of nodes uncovered during the analysis, a "no" edge is added. This method converts Figure 3 to Figure 4.

2.2. Using a NAYO Graph

We say that when a program executes, it starts at the inputs, then runs over the nodes of the NAYO graph. One way to visualize the execution is by growing the trees across the the NAYO graph. Note that as this proof tree grows, it must remain consistent; i.e. it must not contain two nodes connected by a no-edge.

In the testing-as-reachability view described above, testing as just a special case of execution in which:

- We record the tree of paths followed over a NAYO graph.
- We terminate the execution of testing when the search uncovers an interesting node (e.g., a fault).

A *fault explanation tree* is a tree whose leaves are inputs causing the activation of a fault, and whose root is a node containing the fault. Such a fault explanation tree is a subset of the program NAYO graph. Hence, testing is a process of trying to generate fault explanation tree(s) from a NAYO program network. If no such explanation can be generated, then we gain confidence that there are no faults in our program. Of course, as with any testing method, the absence of generated fault explanation trees does not guarantee that the program is actually fault free.

Constructing a tree across a NAYO graph can be a nondeterminate process. For example, consider a search engine exploring Figure 4. This search engine must make choices when faced with options. For example, consider a search engine trying to prove `happy`. Such a search might spread out to reach `tranquillity(hi)` and `healthy`. Which should it explore first? Note that if it explores both, it might have to later choose between `diet(light)` and `diet(fatty)`.

Note also that without the no-edges, a search engine will not face incompatible choices. That is, if the no edges disappear from a NAYO graph, then that system would no longer be necessarily nondeterminate.

3. Average Case NAYO-Graph Reachability

3.1. Defining the Model

Reachability analysis defines an expression for $P[j]$ being the probability that an explanation tree will cover a node at height j given that the number of randomly selected inputs is in. The analysis begins with the following definitions about NAYO graphs such as Figure 4:

- The NAYO graph contains a number of nodes denoted by V. Some fraction of these nodes are and-nodes ($andf$) and the rest are or-nodes (orf). Note that $orf + andf = 1$.
- In the NAYO graph:
  - Or-nodes have orp number of parents (on average).
  - And-nodes have $andp$ number of parents (on average).
  - Or-nodes contradict no number of other or-nodes (on average).
  - No-edges only connect nodes which are found to be incompatible. Hence, and-nodes will never be touched by a no-edge.
- The nodes that connect a set of of inputs to a single node will form a tree. The size of the set of inputs is denoted by in. In that tree:
  - Each and-node has at least one parent that is an or-node.
  - The root of that tree is at height i, where i is the longest path from the root to any leaf (input).
  - The inputs to the system are at height 0. We declare that only or-nodes can be inputs.

- And-nodes will have, on average, *andp* parents in the tree.
- Any or-node at height j $(j > 0)$ will have at least one parent in the tree.
- Up to height j, the tree will contain $n[j]$ nodes.
- At any level, the tree can have up to V nodes, i.e., $n[j] \leq V$.

In the tree, any node at height j $(j > 0)$ will have one parent at height $i = j - 1$ and other parents at height $0 \leq i \leq j-1$. In our simulations, we assume the value of j between 1 and 100. Variable i controls how far back in the graph the node may have its parents:

$$i = \beta(depth) * (j-1) \tag{1}$$

i is the random variable distributed according to the $Beta$ distribution with the mean set to *depth* (0.1 to 0.9 in different simulation runs). As *depth* decreases, parents come from further and further back in the NAYO graph.

To define $P[j]$, we note that a randomly chosen input has odds x that it will stumble across some fault. Further, this input will miss that fault with odds $(1-x)$. If we conduct N random black-box probes, then the odds of a failure not occurring (thus not revealing the fault) is $(1-x)^N$. Hence the probability of finding a fault, hiding in an unknown node within the NAYO graph, in N random tests is Equation 2 (and the inverse is Equation 3):

$$\begin{aligned} p(x,N) &= 1 - \left((1-x)^N\right) && (2) \\ N(p,x) &= log(1-p)/log(1-x) && (3) \end{aligned}$$

Or-nodes are reached at height j via one parent at height $i = j-1$. The probability $P[j]_{or}$ of reaching an or-node at height $j > 0$ is the probability of not missing any of its parents; i.e.

$$P[j]_{or} = 1 - (1 - P\,[j-1]) * \left(\prod_{2}^{orp[j]} (1 - P[i]) \right) \tag{4}$$

Similarly, the probability $P[j]_{and}$ of reaching an and-node at height $j > 0$ is the probability that one of its parents is reached at height $j-1$ and the rest are reached at height $1..(j-1)$; i.e.

$$P[j]_{and} = P\,[j-1] * \left(\prod_{2}^{andp[j]} P[i] \right) \tag{5}$$

The number of parents of an or-node (*orp*) is a random variable distributed according to a gamma distribution $\gamma(\alpha, \frac{\mu}{\alpha})$, where μ is the mean of *orp* (between 1 and 10 in different simulations), and α is its 'skew' (1 to 18 in different simulations). The range of legal values that *orp* may assume is $0 \leq \;\; orp \;\; < \infty$. As α decreases, the distribution becomes narrower, meaning that more or-nodes get the same or a similar number of parents.

Having *and* nodes and *or* nodes, the probability $P[j]$ that a node can be reached at height j is the sum of $P[j]_{and}$ and $P[j]_{or}$, weighted by the frequency of *and* nodes and *or* nodes, i.e.,

$$\begin{aligned} P[j] &= andf * P[j]_{and} + orf * P'[j]_{or} && (6) \\ P'[j]_{or} &= P[j]_{or} * P[j]_{no\,loop} * P[j]_{no\,clash} && (7) \end{aligned}$$

$P'[j]_{or}$ is similar to the original $P[j]_{or}$, but it is modified by Equation 7, for the following reasons. Recall that or-nodes can contradict, on the average no other *or* nodes. The positive value of no variable implies that there is nondeterminacy in the model. When $no > 0$, the inference engine will need to choose between competing paths from time to time. no is a random variable which follows gamma distribution with mean no_μ $(0, 1 \ldots 4)$ and skew no_α $(1, 2, 4 \ldots 18)$. The probability $P[j]_{no\ clash}$ that a new node can be added to a NAYO path of size $n[j]$ at level j is the probability that this new node will not contradict any of the or-nodes in the current explanation tree:

$$P[j]_{no\,clash} = \left(1 - \frac{no}{V}\right)^{n[j]*orf} \tag{8}$$

In this equation, V is the number of nodes in the whole graph (simulated as $1000, 2000, \ldots 1,000,000$).

Not only must a new node not contradict with other nodes in the explanation tree, it must also not introduce a loop into the tree, since loops do not contribute to revealing unseen nodes.

$$P[j]_{no\,loop} = \left(1 - \frac{1}{V}\right)^{n[j]*orf} \tag{9}$$

Observe the use of $n[j] * orf$ in Equation 8 and Equation 9. And-nodes contradict no other nodes; hence we we only need to consider contradictions for orf of the system. Also, since every and-node has an or-node as a parent, then we need only check for loops amongst the or-nodes.

Finally, we offer some initial conditions. in is the number of inputs to our system. Inputs are represented by or nodes, since and-nodes always have pre-conditions. Hence:

$$\begin{aligned} P[0]_{and} &= 0 && (10) \\ P[0]_{or} &= \frac{in}{V} && (11) \end{aligned}$$

The above reachability model was run 100,000 times using values for all model parameters randomly selected from the ranges described in this section. The results are shown below.

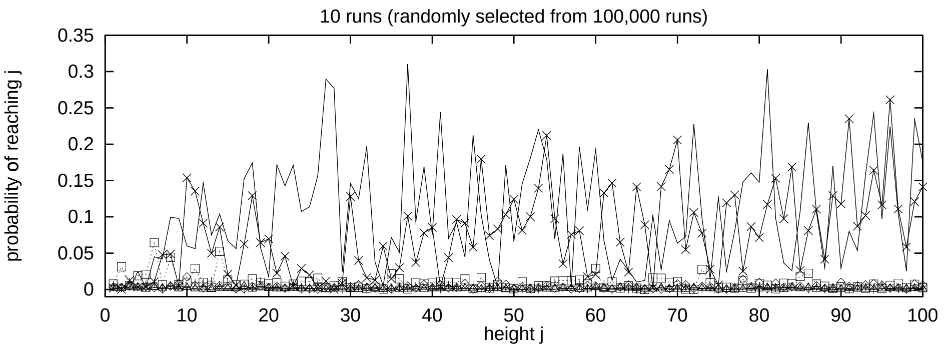

Figure 5. Some randomly selected runs from the reachability model.

3.2. Model Output

As we might expect, the behaviour of our reachability model is quite variable: see the ten randomly selected runs shown in Figure 5. Such variability is to be expected: the model contains many random variables such as $orp, andp, andf, no, i$

However, significant regularities can be seen if we examine numerous runs. Recall that the number of tests N required to be 99% sure of reaching a node with probability of reaching $P[j]$ can be calculated from Equation 3 using $x = 0.99$ and $p = P[j]$. Some frequency distributions of the calculated N are shown in Figure 6. Note two important effects:

The in effect: For shallow probes, the number of inputs crucially determines reachability. For example, at $j = 10$ and $in = 50$, 1,000,000 random are needed to reach 50% of the program; see Figure 6.A. However, at $j = 10$ and $in = 1000$, 10,000 random tests suffice to reach 50% of the program; see Figure 6.B.

The j effect: As we probe deeper into the program (increasing j), more and more of the program can be reached by a relatively small number of randomly selected inputs. For example, by $j = 90$ (Figure 6.F), nearly half of the program can be reached using 100 randomly selected inputs,

The in effect has been reported previously in the literature by Rothermel et.al. [18]. In their experiments, reducing test suite size significantly decreased the number of faults detected. The same effect can be seen in our simulations outputs. As the input size is increased from Figure 6.A to Figure 6.B, the nodes reached by 10,000 tests increases from 10% to 50%. Rothermel et.al. did not study the j effect. Our results strongly suggest that the j effect dominates the in effect. By $j = 90$, increasing the input size by a factor of 20 has little effect on the percentage of the system reached by 100 random inputs (40% in Figure 6.E vs 50% in Figure 6.F). Note that systems have to be very small indeed to only support $j < 10$.

One interesting feature is that for a wide range of NAYO graphs, most of those graphs are reachable using a small number of randomly generated inputs *despite* NAYO computation being nondeterminate. A recent literature review

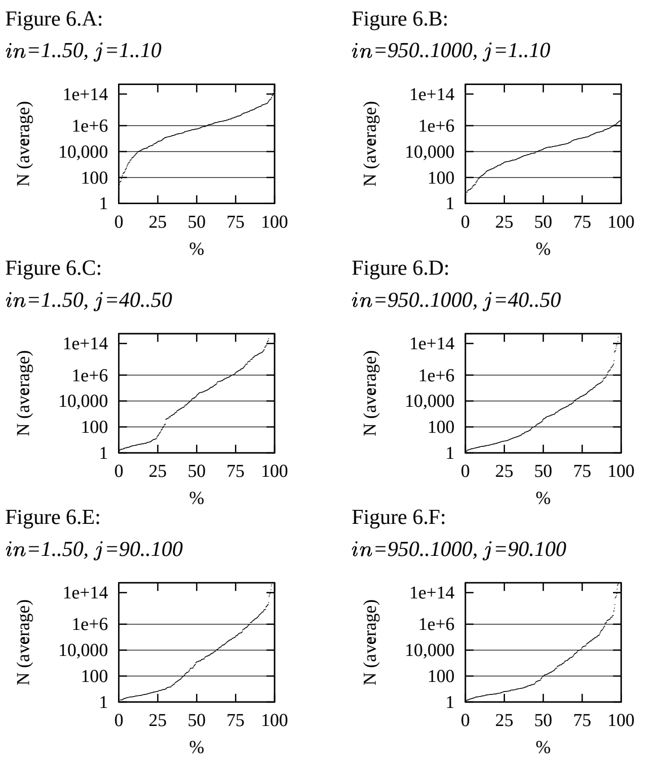

Figure 6. Outputs from the reachability model, restricted to certain ranges. Y-axis comes from Equation 3; X-axis generated by sorting simulation outputs. Each plot represents 400 simulations using randomly selected parameters for the reachability model.

offers much evidence that this is a commonly observed effect [12]. For example, researchers in AI and requirements engineering explore inconsistent and nondeterminate theories. A repeated result is that committing to a randomly selected resolution to a conflict reaches as much of a program as carefully exploring all resolutions to all conflicts [7, 13, 14, 21]. This is consistent with the search space within our programs containing many paths to the same point- a view very consistent with Figure 6.

Other results from software engineering and knowledge engineering literature suggest that the effects derived from the reachability model have been widely observed. An often repeated observation is that a small number of inputs can often reach significant errors in a program [4, 12]. Various researchers have noted that the portions of a program used in normal operation are a small subset of the total program [3, 6], and that test coverage suites often do not target the entire program [8, 11]. A repeated observation in classical mutation testing literature is that most program mutants generate the same behaviour [2, 5, 15, 22].

These empirical observations are consistent with the hypothesis that programs include easily reachable and very unreachable zones. The same effect can be seen in our simulation outputs. Recall that the y-axis of Figure 6 is a logarithmic scale: as our curves rise on that scale, they are escalating into very unreachable zones.

4. Nondeterminism and Maintenance

Based on the above results and literature review, we argue that a large class of nondeterminate systems can be probed with a small number of random inputs. Hence, we are optimistic about our ability to quickly test nondeterministic AI systems and requirement models.

However, we are more cautious regarding the maintenance of nondeterministic systems. Figure 6 might give the impression that the transitions between easily and non-easily testable systems is quite gradual. As we shall see, this is not the case.

In studies with decision trees learnt from the simulation outputs, we have learnt that small changes to a system can dramatically alter the reachability. In those studies, each of the 100,000 runs of the reachability model were classified as follows. Firstly, we calculated P_{av}, i.e. the average probability of reaching the node at any depth within the graph:

$$P_{av} = \frac{\sum_{j=0}^{100} P[j]}{100}. \qquad (12)$$

Next, the P_{av} figures were converted to number of required tests using Equation 3 and classified as follows:

$$Type = \begin{cases} fast\,and \\ cheap & if\ N(0.99, P_{av}) < 10^2, \\ \\ fast\,and \\ moderately \\ expensive & if\ N(0.99, P_{av}) < 10^4, \\ \\ slow, and \\ expensive & if\ N(0.99, P_{av}) < 10^6, \\ \\ impossible & otherwise. \end{cases}$$

For example, testing a particular NAYO graph is classified as "fast and cheap" if we require less than 100 random tests to be 99% sure of reaching all of the graph. Finally, to build the decision trees, we used an automatic machine learner (C4.5 [17]) to generate Figure 7. The learner took as input the 100,000 inputs classified using the $Type$ define above. The learner generated as output a tree correlating the values of simulation parameters introduced earlier in this section, with the outcome of the test classifications (the estimated error of the tree on unseen cases is 28.3%).

The learnt tree clearly shows the *cliffs of reachability*: thresholds where the reachability of a system can change dramatically. One such cliff can be seen in Figure 8 which contrasts two pathways through Figure 7. The "A" pathway requires 10^4 to 10^6 tests to be 99% sure we can probe all of the graph. The "B" pathway requires $<$ 100 tests to be 99% sure we can probe all of the graph. Suppose a NAYO system falls into the "B" pathway; i.e. it is easily testable. Suppose further that, during maintenance, the average number of and-node parents in that graph changes from (e.g.) 3.75 to 4.25. This single change could imply that the graph switches from the "B" path to the "A" path; i.e. suddenly that graph would require at least tens of thousands more tests to be 99% sure we can probe all of the graph.

5. Discussion

This article rejects that traditional view that the behaviour of nondeterminate systems makes them too unpredictable to be testable. Based on a mathematical analysis of NAYO graph reachability and a literature review [12], we have argued that wide range of nondeterminate systems have easy reachability; i.e. they can be adequately and quickly probed with a small number of randomly selected tests.

However, while optimistic regarding the testability of nondeterministic systems, we are more cautious regarding their maintainability. Small changes to a system can suddenly convert an easily testable system to a very system that is very difficult to test. We conclude that a nondeterministic

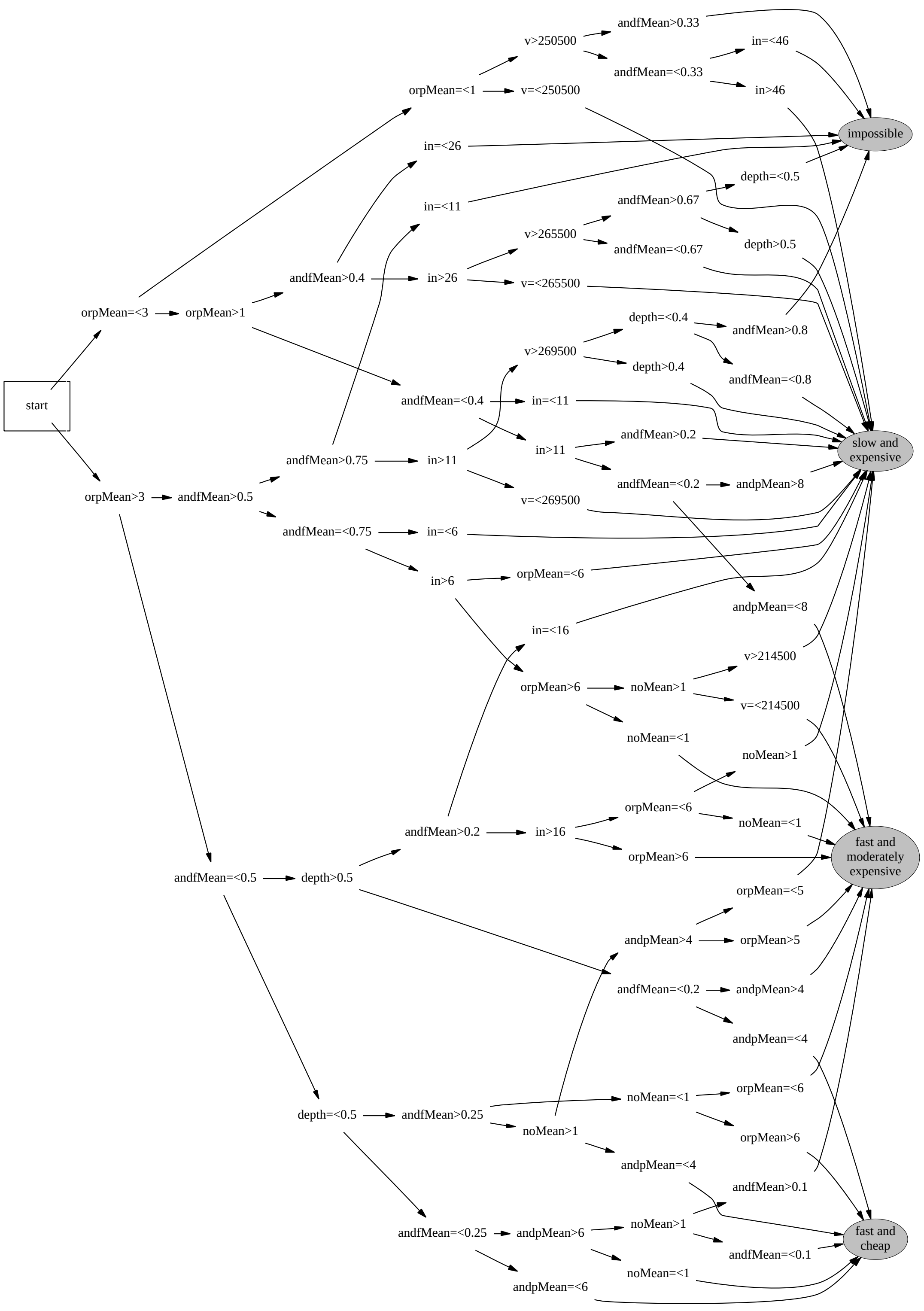

Figure 7. Classifications seen from 100,000 runs of the reachability model. Tree generated using the C4.5 algorithm [17].

	Graph	
	A	B
	slow and expensive	fast and cheap
	(requires $10^4..10^6$ tests)	(requires $< 10^2$ tests)
orp_μ	> 3	> 3
$andf_\mu$	$0.2 \ldots 0.5$	$0.2 \ldots 0.5$
in	≤ 16	any
no_μ	any	> 1
$andp_\mu$	any	≤ 4

Figure 8. Small changes in a NAYO graphs may imply a large change in the number of required tests.

system may require retesting after seemingly trivial changes to the system.

Our analysis also shows that testing nondeterminate systems is not necessarily more difficult that testing deterministic ones. Figure 7 shows a complex interaction of factors that determine the testability of a system. The critical factors are shown near the root of the tree. Note that the the mean number of no-edges is not critical. Other factors, such as the frequency of the and-nodes are more critical. That is, factors relating to the overall structure of the program may be more critical than nondeterminism in determining overall system testability.

We are very enthusiastic about the potential this framework has for software testing. This is a unifying framework, bridging the differences between software paradigms used during the development. In fact, it is suitable for testing early life-cycle artifacts as well as programs ready for release. The degree to which the NAYO graph reveals its nodes may be used as a testability measure. Different programs will have different testabilities. Testers, for example, may want to attribute a higher degree of confidence to the testing results of a testable program because it hides less [1]. Since revealing nondeterminacy is one of the goals of this framework, the probability that it will go unnoticed is reduced. This has the potential for increasing the reliability.

The precondition for investigating any of these topics is the development of an automated environment for building NAYO graphs from the reasonable set of traditional program representations. Without such a tool, we can only discuss the general properties of classes of programs (e.g. the discussion seen in this article). With such a tool, we will be able the specific reachability, hence testability, of a specific program. We are currently building such a tool for the specification language (PROMELA) and the logic programming language (Prolog).

Acknowledgements

The comments of the anonymous reviewers lead to a significant simplification and clarification of this paper. Also, thanks to Helen Burgess for a timely review of the final draft of this paper. This work was partially supported by NASA through cooperative agreement #NCC 2-979.

References

[1] L. S. A. Bertolino. On the use of testability measures for dependability assessment. *IEEE Trans. Software Engineering*, 22(2):97–108, Feb 1996.

[2] A. Acree. *On Mutations*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1980.

[3] A. Avritzer, J. Ros, and E. Weyuker. Reliability of rule-based systems. *IEEE Software*, pages 76–82, September 1996.

[4] J. Bieman and J. Schultz. An empirical evaluation (and specification) of the all-du-paths testing criterion. *Software Engineering Journal*, 7(1):43–51, 1992.

[5] T. Budd. *Mutation analysis of programs test data*. PhD thesis, Yale University, 1980.

[6] R. Colomb. Representation of propositional expert systems as partial functions. Artificial Intelligence (to appear), 1999. Available from `http://www.csee.uq.edu.au/~colomb/PartialFunctions.html`.

[7] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI '94*, 1994.

[8] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Press, 1997.

[9] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specification. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[10] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[11] J. Horgan and A. Mathur. Software testing and reliability. In M. R. Lyu, editor, *The Handbook of Software Reliability Engineering*, pages 531–565, McGraw-Hill, 1996.

[12] T. Menzies and B. Cukic. Adequacy of limited testing for knowledge based systems. *International Journal on Artificial Intelligence Tools (IJAIT)*, June 2000. (to appear).

[13] T. Menzies, S. Easterbrook, B. Nuseibeh, and S. Waugh. An empirical investigation of multiple viewpoint reasoning in requirements engineering. In *RE '99*, 1999. Available from `http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-009.pdf`.

[14] T. Menzies and C. Michael. Fewer slices of pie: Optimising mutation testing via abduction. In *SEKE '99, June 17-19, Kaiserslautern, Germany. Available from `http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-007.pdf`*, 1999.

[15] C. Michael. On the uniformity of error propagation in software. In *Proceedings of the 12th Annual Confererence on Computer Assurance (COMPASS '97) Gaithersburg, MD*, 1997.

[16] K. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and J. W. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Trans. on Software Engineering*, 18(1):33–44, Jan 1992.

[17] J. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[18] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimizaton on the fault-detection capabilities of test suites. In *Proceedings of International Conference on Software Maintenance '98*, pages 34–43, November 1998. Available from `http://www.cis.ohio-state.edu/~harrold/research/webpapers/icsm98-min.pd%f`.

[19] F. Schneider, S. Easterbrook, J. Callahan, G. Holzmann, W. Reinholtz, A. Ko, and M. Shahabuddin. Validating requirements for fault tolerant systems using model checking. In *3rd IEEE International Conference On Requirements Engineering*, 1998.

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

[21] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings, AAAI '96*, pages 971–978, 1996.

[22] W. Wong and A. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185–196, December 1995.