# You seem friendly, but can I trust you?

Tim Menzies, David Owen, Bojan Cukic

Lane Department of Computer Science
West Virginia University
PO Box 6109 Morgantown, WV 26506-6109, USA
tim@menzies.com {dowen|cukic}@csee.wvu.edu

**Abstract.** A "stranger agent" is an agent you did not build but may need to work with. Such strangers may come with a certificate saying "I have been tested, trust me". In the special case of nondeterministic agents built via FSMs, the level to which we can trust this certificate can be determined via a simple summary of topological features of the agent.

## 1 Introduction

*A stranger is someone that you don't know.*
*Most strangers will not try to harm you,*
*but some strangers are dangerous,*
*even if they dress nice or look friendly.*
*Never take rides, candy, gifts, or money from strangers.*
*It's okay to say "NO THANK YOU."*
*– "Stranger Danger"*[1]

Two major trends in software engineering is the increased used of COTS (commercial-off-the-shelf) products and decentralized processing. In the near future, applications that work via the coordinated effort of many components, most of which we don't build ourselves but buy (or hire) from others. Can we trust such *stranger agents* to deliver the functionality we require?

To make this analysis interesting, we make five assumptions:

**Nondeterminacy:** The stranger agents may be nondeterministic. For example, agents conduct searches of complicated spaces may use heuristics to direct their processing. These heuristics may use random choice to (e.g.) break ties within the reasoning.

**Safety:** Somewhere within our collection of collaborating agents is a safety critical agent we are very concerned about. This assumption motivates us to make as detailed an analysis as possible of stranger agents.

**Certified:** Each new agent in our community arrives with a certificate saying "I have been thoroughly tested" (imagine the firm handshake of the smiling used-car salesman). If we believe that certificate, then we might test other agents in our community before testing this agent.

[0] Second NASA Goddard Workshop on Formal Approaches to Agent-Based Systems "FAABS II", October 2830, 2002 Greenbelt, Maryland, USA. http://fmw.gsfc.nasa.gov. Date April 14, 2003. Wp ref: faabs2/taming. Url: http://menzies.us/pdf/02trust.pdf

[1] http://www.ci.mesa.az.us/police/stranger.htm

**Benevolence:** The agents we are assessing are not deliberately malicious agents. Our goal in assessing stranger agents is to check that the benevolent author of some other agent did not inadvertently miss something during their testing.

**Need lightweight assessment:** If the agents are COTS products, we must assume that they are true strangers; i.e. we can't examine them in great detail since such an examination may violate corporate confidentiality. Hence, any assessment we make of agent must be a *lightweight* assessment that does not require or reveal specifics of the internal of an agent.

How suspicious should we be of nondeterministic certified benevolent agents which we can't examine in detail and which are participating in an community of agents containing a safety critical agent? Can we assess our level of doubt? Is possible to assess how hard or easy it is to test a nondeterministic agent and, therefore, how much or little we should believe in those certificates? And can all the above be done in a *lightweight* manner so as to not intrude on the privacy of our agents?

While in the general case this assessment is difficult, for one definition of "testability" and for nondeterministic agents constructed from communicating finite state machines FSMs, we show here that that the assessment is simple. Further, since the assessment just relies on high-level summaries of FSM topology, it is *lightweight* in the above sense of the term.

The interesting point of this analysis is that, according to classical software reliability theory, it is impossible. For example Nancy Leveson says that "nondeterminism is the enemy of reliability" [5]. We must disagree. Like it or not, future software systems will be based on collaborations of distributed units, each with their own style of reasoning which may include nondeterministic inference. We demonstrate in this paper that we need not necessarily fear this future since nondeterministic agents can be tested and the probability that those tests will reveal errors can be determined from the topology of the agents.

The rest of this paper is structured as follows. After defining FSMs, we describe LURCH1, our *nondeterministic simulator*. This is followed by a definition of *testability* used for our study. We next describe a *model mutator* that can generate many FSMs and the TAR2 *treatment learner* that found the topological features of the FSMs that makes them more or less testable.

## 2 FSMs

Our analysis assumes agents are implemented or can be modelled as FSMs. We therefore start by defining an FSM.

An FSM has the following features:

- Each FSM $M \in S$ is a 3-tuple $(Q, \Sigma, \delta)$.
- $Q$ is a finite set of states.
- $\Sigma$ is a finite set of input/output symbols.
- $\delta : Q \times B \longrightarrow Q \times B$, where $B$ is a set of zero or more symbols from $\Sigma$, is the transition function.

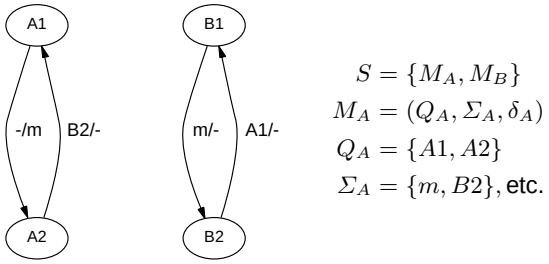



**Fig. 1.** A system of communicating FSMs ("m" is a message passed between the machines).

Figure 1 shows a very simple communicating FSM model. States are indicated by labelled ovals, and edges represent transitions that are triggered by input and that result in output. Edges are labelled: *input* / *output*. An important distinction in Figure 1 is ebtween *consumables* and *non-consumables*. A transition triggered by a message *consumes* the message, so that it is no longer able to trigger another. But states are unaffected by transitions they trigger; they are good for an arbitrary number of transitions.

FSMs can be characterized via the following parameters:

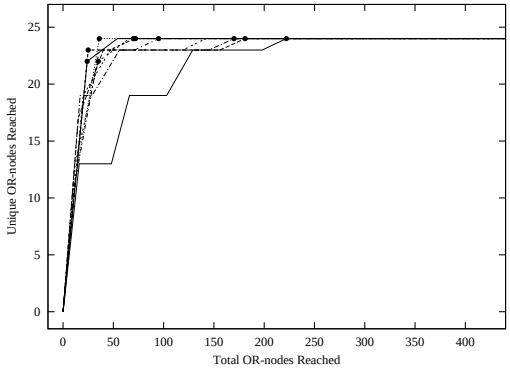
1. The number of individual finite-state machines in the system. Figure 1 has two.
2. The number of states per finite-state machine. Figure 1 has two states per mission (true and false).
3. The number of transitions per machine. Figure 1 has two transitions per machine.
4. The number of inputs per transition that are states in other machines. Figure 1 has two such inputs: (*A2*, *B2*).
5. The number of unique *consumable* messages that can be passed between machines. Figure 1 has one such message: *m*.
6. The number of inputs per transition that are consumable messages. Figure 1 uses *m* as input in one transition.
7. The number of outputs per transition that are consumable messages. In Figure 1, *m* appears as an output in one transition.

## 3 Nondeterminism

To model nondeterminism in this approach, we will assume that the LURCH1 inference engine is being used to process the FSMs. To use LURCH1, models in different representations are partially evaluated into variant of a directed and-or graph. Conceptually, this graph is copied for $N$ time ticks and the outputs generated at time $i-1$ become inputs for time $i$. At runtime, LURCH1 maintains a *frontier* for the search. When a node is popped off the frontier, it is discarded if it contradicts an assertion made at the same time. Otherwise, the node is added to the list of assertions.

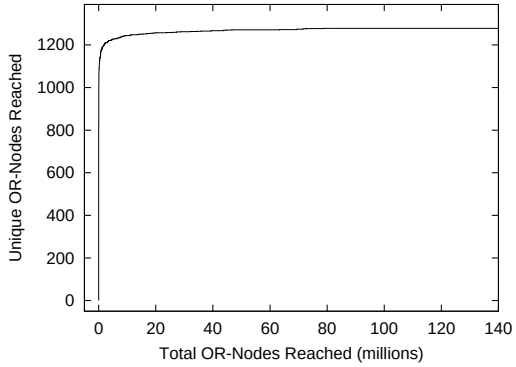
LURCH1's nondeterministic nature arises from how the search proceeds after a new assertion is made. If all the pre-conditions of the descendants of the new assertions have

been asserted then these descendants are added to the frontier *at a random position*. As a result, what is asserted at each run of LURCH1 can differ. For example, if the node for $x$ and $\neg x$ are both reachable from inputs, they will be added to the frontier in some random order. If $x$ gets popped first, then the node $x$ will be asserted and the node $\neg x$ will be blocked. But if the node $\neg x$ gets popped first, then the node $\neg x$ will be believed




Random search results for model of Dekker's solution to the two-process mutual exclusion problem (the model comes from Holzmann [4]). Dots show when an error added to the model is found by the search. The error is found in every case.

**Fig. 2.** Random search of AND-OR graphs representing FSM models is effective in finding errors.




Random search results for a very large randomly generated FSM model, for which the set of FSMs being studied would require at most $2.65 \times 10^{178}$ states.

**Fig. 3.** Random search of AND-OR graphs is scalable to very large models.

and the node $x$ will be blocked.

The random search of LURCH1 is theoretically incomplete but, in practice, it is surprisingly effective. For example, Figure 2 (from Menzies et.al. [8]) shows ten trials with a LURCH1 search over a model of Dekker's solution to the two-process mutual exclusion problem (the original model comes from Holzmann [4]). The dots represent an error added to the model and found quickly by random search in all ten trials. LURCH1 is very simple, yet can handle searches much larger than many model checkers. For example, Figure 3 shows random search results for a very large FSM model. The composite FSM representing all interleavings of the individual machines in the Figure 3 model would require at most $2.65 \times 10^{178}$ states. This is well beyond the capability of model checking technology ($10^{120}$ states according to [3]).

## 4 Testability

The claim of this paper is that it is possible to identify agent properties that predict for "testability". This section describes our definition of "testability".

Note the *plateau* shape of Figure 2 and Figure 3. If some method can increase the height of that plateau, then that method would have increased the chances the odds of finding a defect.

This definition of increased "testability" is a reasonable model-based extension of standard testability definitions. According to the IEEE Glossary of Software Engineering Terminology [1], testability is defined as "the degree to which a system of components facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met". Voas and Miller [9], and later Bertolino and Stringini [2] clarify this definitions, arguing that testability is "the probability that the program will fail under test if it contains at least one fault". If LURCH1 quickly reveals many unique reachable nodes in the model quickly and if some of these nodes contain faulty logic, then those faults must be exposed.

Note that when the search reaches a plateau, there are no guarantees provided about failure free field operation. But, unvisited nodes in the system model are difficult to reach in the operational environment too, hence the operational failure probability due to testable design of the model does not increase.

## 5 Model Mutator

In the study shown below, a nondeterministic inference process (LURCH1) was run over 15,000 FSMs generated semi-randomly. The testability of each run was assessed via the percentage of FSM nodes reached in each run. This section describes how the 15,000 FSMs were generated.

Each FSM had parameter values drawn at random from the following ranges:

1. 2–20 individual FSMs.
2. 4–486 states (states within all within machines).
3. 0–272 transitions per machine.
4. 0–737 transition inputs that are states in other machines.

5. 0–20 unique consumable messages.
6. 0–647 transition inputs that are consumable messages.
7. 0–719 transition outputs that are consumable messages.

These parameters where selected to ensure that FSMs from real-world specifications fell within the above ranges (for details of those real-world models, see [8]).

The FSM generation process not truly random. Several *sanity checks* were imposed to block the generation of bizarre FSMs:

- The *current state* and *next state* must come from the machine in which the transition is defined and must not match.
- Inputs that are states must come from *other* machines, and none may be mutually exclusive (the transition could never occur if it required mutually exclusive inputs).
- The set of inputs that are messages from other machines contains no duplicates.
- The set of outputs that are messages to other machines contains no duplicates.

## 6 Data Mining

Having generated 15,000 outputs, some data mining technology is required to extract the essential features of all those runs. The data miner used in this paper was the TAR2 *treatment learner* [6]. This is a non-standard data miner, so we take care to fully introduce it here.

The premise of treatment learning, and the reason why we use it, is that the learnt theory must be *minimal*. TAR2 was an experiment in generating the essential minimal differences between classes. To understand the algorithm, consider the log of golf playing behavior seen in Figure 4. In that log, we only play *lots* of golf in $\frac{6}{5+3+6} = 43\%$ of cases. To improve our game, we might search for conditions that increases our golfing frequency. Two such conditions are shown in the `WHERE` test of the select statements in Figure 4. In the case of `outlook=overcast`, we play *lots* of golf all the time. In the case of `humidity` $\leq$ `90`, we only play *lots* of golf in 20% of cases. So one way to play lots of golf would be to select a vacation location where it was always overcast. While on holidays, one thing to watch for is the humidity: if it rises over 90%, then our frequent golf games are threatened.

The tests in the `WHERE` clause of the select statements in Figure 4 is a *treatment*. Classes in treatment learning get a score and the learner uses this to assess the class frequencies resulting from *applying a treatment* (i.e. using them in a `WHERE` clause). In normal mode, TAR2 does *controller learning* that finds a treatment which selects for better classes and reject worse classes By reversing the scoring function, treatment learning can also select for the worse classes and reject the better classes. This mode is called *monitor learning* since it finds the thing we should most watch for. In the golf example, *outlook = 'overcast'* was the controller and $humidity \geq 90$ was the monitor.

TAR2 automatically explores a very large space of possible treatments. TAR2's configuration file lets an analyst specify a search for the best treatment using conjunctions of size 1,2,3,4, etc. Since TAR2's search is elaborate, an analyst can automatically find the *best* and *worst* possible situation within a data set. For example, in the golf example, TAR2 explored all the attribute ranges of Figure 4 to learn that the *best* situation was *outlook = 'overcast'* and worst possible situation was $humidity \geq 90$.

TAR2 also comes with a *N-way cross validation* tool that checks the validity of a select statement. In this process, a training set is divided into $N$ buckets. For each bucket in turn, a treatment is learned on the other $N - 1$ buckets then tested on the bucket put aside. A treatment is preferred if it is *stable*; i.e. works in the majority of all $N$ turns.

Theoretically, TAR2 is intractable since there are an exponential number of possible attribute ranges to explore. TAR2 culls the space of possible attribute ranges using a heuristic *confidence1* measure that selects attribute ranges that are more frequent in

| *outlook* | *temp(°F)* | *humidity* | *windy?* | *class* |
|---|---|---|---|---|
| *sunny* | *85* | *86* | *false* | *none* |
| *sunny* | *80* | *90* | *true* | *none* |
| *sunny* | *72* | *95* | *false* | *none* |
| *rain* | *65* | *70* | *true* | *none* |
| *rain* | *71* | *96* | *true* | *none* |
| *rain* | *70* | *96* | *false* | *some* |
| *rain* | *68* | *80* | *false* | *some* |
| *rain* | *75* | *80* | *false* | *some* |
| *sunny* | *69* | *70* | *false* | *lots* |
| *sunny* | *75* | *70* | *true* | *lots* |
| *overcast* | *83* | *88* | *false* | *lots* |
| *overcast* | *64* | *65* | *true* | *lots* |
| *overcast* | *72* | *90* | *true* | *lots* |
| *overcast* | *81* | *75* | *false* | *lots* |

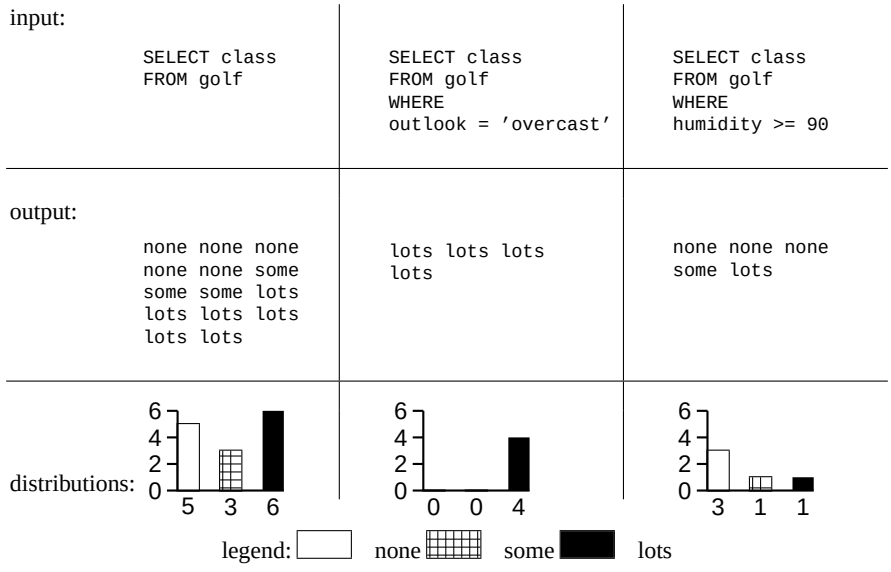


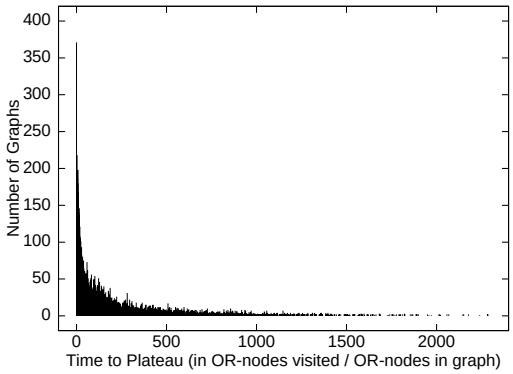

**Fig. 4.** Class distributions selected by different conditions.

good classes than in poorer classes (for full details, see [7]). The use of *confidence1* has been quite successful: TAR2's theoretically intractable nature has yet to be of practical concern.

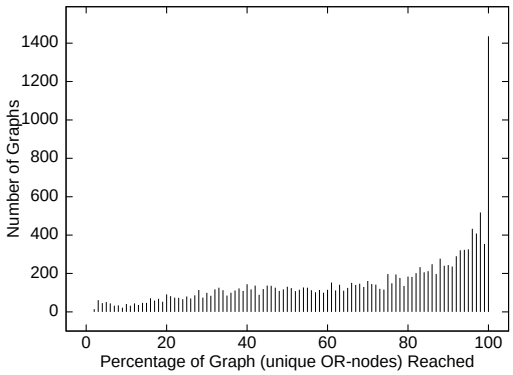
## 7 Experiments

We now have all the pieces required for our study. The model mutator can generate many FSMs, LURCH1 can nondeterministically execute them, and TAR2 can find the FSM features that change testability (plateau height).

Figure 5 shows a summary of LURCH1 executing over 15,000 semi-randomly generated models. The top histogram summarizes *time-to-plateau* results, e.g., *time-to-plateau* for approximately 375 models was 0 ($< 1\times$ graph size). The average value




Average time-to-plateau = 208.0 $\times$ NAYO size (NAYO size is polynomial in the size of FSM model input).




Average plateau height = 69.39%.

**Fig. 5.** Summary of time-to-plateau (top) and plateau height (bottom) results for 15,000 models.

was about $208\times$ graph size. The right side of the plot shows that, for a few models, nearly 2,500 $\times$ the size of the graph was processed before a plateau was reached. This may seem like a lot, but compared to the exponential size of the composite FSM exhaustively searched by a model checker, a factor of 2,500 is insignificant. The bottom part of Figure 5 is a histogram summarizing search *plateau height* for our 15,000 semi-randomly generated models. The average value was about $70\%$, with a significant number of models showing much lower plateaus.

The top part of Figure 5 indicates that plateaus were reached quickly for nearly all models, whether high or low plateaus. So the key distinction, in terms of testability, is plateau height. We would like to how FSM models yielding high search plateaus are different from FSM models yielding low search plateaus. Specifically, what ranges of the attributes listed above (number of machines, number of states, etc.) characterize the models with high plateaus represented by the right side of the bottom histogram in Figure 5?

In our first simple experiment we used TAR2 to determine what single attribute, and what range of that attribute, could most significantly constrain our models to high plateaus (just like the very simple TAR2 golf example in the previous section, where we found that restricting *outlook* to *overcast* led to *lots* of golf). TAR2 suggested the following treatment: restrict *state inputs* to its highest range (590–737). To understand what that means, consider Figure 6, which shows the number of *state inputs* vs. plateau height (with a dot for each model). On the left, where there are few *state inputs*, we see plateau height distributed all the way from about $5\%$ to $100\%$. But further to the right, where the number of *state inputs* is high, we see only high plateaus—once *state inputs* exceeds 1,000 we see only plateaus over $60\%$. So TAR2's suggested treatment, that we restrict *state inputs* 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 7 shows a summary of results from a series of experiments, in which we tried to determine which combinations

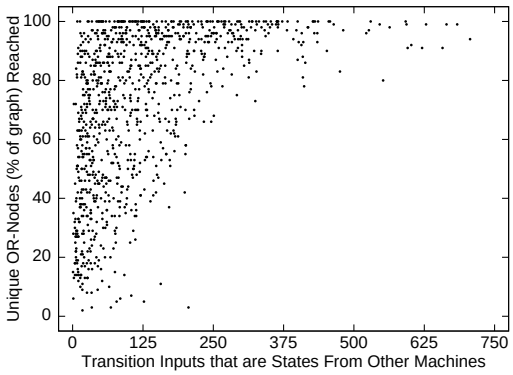



**Fig. 6.** The number of transition inputs that are states from other machines vs. plateau height.

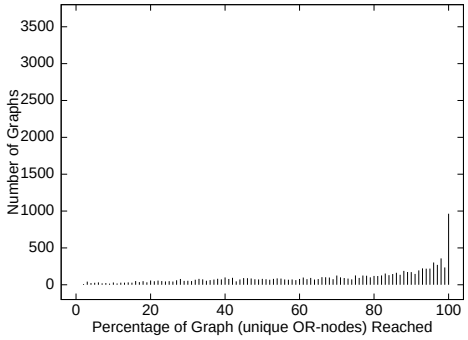
of attribute ranges (each treatment considers 4 attributes) are favorable for testability and which give us very untestable graphs. Surprisingly, the three top parameters are low for not only highly testable graphs, but also for graphs that are very difficult to test (the number of finite-state machines and the total number of states are more significant than the total number of transitions). So if we restrict our sample to simpler models (fewer machines, fewer states, fewer transitions) the testability results are polarized.

The bottom half of Figure 7 shows which attributes have the greatest affect on testability, given that the top three are held low. The most significant attribute is *state inputs*, followed by *message inputs* and *message outputs*. To verify the result from TAR2, we need to make sure that the treatments learned apply generally, not just to the data from the original experiment. Figure 8 shows a comparison of plateau height (our indicator of testability) for the original data (left) and a new 10,000 input models (right) generated using TAR2's recommendation of what treatment most improves plateau height; i.e.

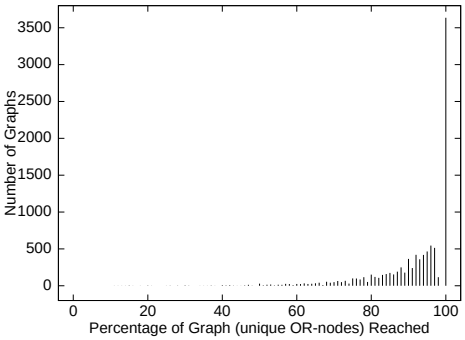
| | ⟵ Better Treatments | | |
|---|---|---|---|
| Machines | lowest (2–4) | lowest | lowest |
| States | lowest (4–49) | lowest | lowest |
| Transitions | low (0–109) | low | low |
| State Inputs | high (443–737) | | |
| Messages<br>Message Inputs | (not significant)<br>high (389–647) | | |
| Message Outputs | | | high (432–719) |

| | Worse Treatments ⟶ | | |
|---|---|---|---|
| Machines | lowest (2–4) | lowest | lowest |
| States | lowest (4–49) | lowest | lowest |
| Transitions | lowest (0–54) | lowest | lowest |
| State Inputs | | | lowest (0–147) |
| Messages<br>Message Inputs | (not significant)<br>lowest (0–129) | | |
| Message Outputs | lowest (0–143) | | |

**Fig. 7.** Best and worst treatments learned by TAR2.




Original search data (i.e. Figure 5 with an adjusted scale that matches the new search data plot shown immediately below)—average plateau height = 69.39%.




Search data for input models generated according to TAR2's suggestions—average plateau height = 91.34%.

**Fig. 8.** Comparison of plateau height for original search data (top) and new data based on TAR2's suggested treatments.

1. 2–5 FSMs.
2. 4–49 states.
3. 0–43 transitions.
4. 0–247 transition inputs that are states from other machines.
5. 0–10 unique consumable messages.
6. 0–229 transition inputs that are consumable messages.
7. 0–241 transition outputs that are consumable messages.

Figure 8 shows that we can meet the goal specified in the introduction. It is possible to learn parameters of an FSM that significantly improve FSM testability. In this case the improvement was a change in the average plateau height from 69% to 91%.

## 8 Discussion

In the case of FSMs via a nondeterministic algorithm and assessed via plateau height, we have applied our analysis method to automatically learn the features that most effect FSM testability via model mutators and TAR2.

We believe that this method would generalize to other representations and other definitions of testability. The only essential requirement for such a study is the availability of an automatic oracle of success. With such an oracle available, then mutation plus treatment learning can find model features that select for successful runs.

Another possibility that arises from this work is that we can identify design parameters that make our nondeterministic FSM-based agents *more* or *less* testable. For example, given two implementations of the same requirement, we could favor the implementation that results in a more testable system. That is, we can *design for testability*,

even for nondeterministic agents.

## References


1. IEEE glossary of software engineering terminology, ANSI/IEEE standard 610.12, 1990.
2. L. S. A. Bertolino. On the use of tesability measures for dependability assessment. *IEEE Transactions on Software Engineering*, 22(2):97–108, 1996.
3. E. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
4. Gerard J. Holzmann. Basic SPIN Manual. Available at `http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.htm`.
5. N. Leveson. *Safeware System Safety And Computers*. Addison-Wesley, 1995.
6. T. Menzies and Y. Hu. Constraining discussions in requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://menzies.us/pdf/01lesstalk.pdf`.
7. T. Menzies and Y. Hu. Agents in a wild world. In C. Rouff, editor, *Formal Approaches to Agent-Based Systems, book chapter*, 2002. Available from `http://menzies.us/pdf/01agents.pdf`.
8. T. Menzies, D. Owen, and B. Cukic. Saturation effects in testing of formal models. In *ISSRE 2002*, 2002. Available from `http://menzies.us/pdf/02sat.pdf`.
9. J. Voas and K. Miller. Software testability: The new verification. *IEEE Software*, pages 17–28, May 1995. Available from `http://www.cigital.com/papers/download/ieeesoftware95.ps`.


This article was processed using the LaTeX macro package with LLNCS style