

What Makes Finite-State Models more (or less) Testable?¹

David Owen, Tim Menzies, Bojan Cukic
Lane Department of Computer Science
West Virginia University
PO Box 6109 Morgantown, WV 26506-6109, USA
{downen|cukic}@csee.wvu.edu, tim@menzies.com

Abstract

Finite-state machine (FSM) models are commonly used to represent software with concurrent processes. Established model checking tools can be used to automatically test FSM models, but this approach can be very resource-intensive and may not scale to larger models.

In this paper we use a partial random search technique for testing FSMs. Such a search has been shown previously to be a surprisingly effective and scalable to very large models. Random search is also very fast and so can be used to gather testability data representing a wide range of FSM models. The TAR2 treatment learner can then summarize that data to determine what FSM attributes characterize models that are easiest to test.

KEYWORDS: Reasoning techniques, software specification, testing, requirements engineering.

1 Introduction

How should we test software? Given a range of possible test methods, when is one technique preferred? Typically, these questions are answered with reference to the inherent properties of the assessment mechanism, e.g., [9, 10]. This paper takes another approach. We find that, if the test method is fixed, we can identify classes of software that are more or less testable.

Such *testability* results are useful when judging different design options or estimating the expected testing effort for a piece of software. Also, testability results might help us decide when to *change* test methods, e.g., when the model under consideration falls outside of the zone of competency of a particular test method. Further, it may be possible to manipulate models to make them more testable without sig-

nificantly changing the implementation they represent. This is an exciting possibility, though highly speculative.

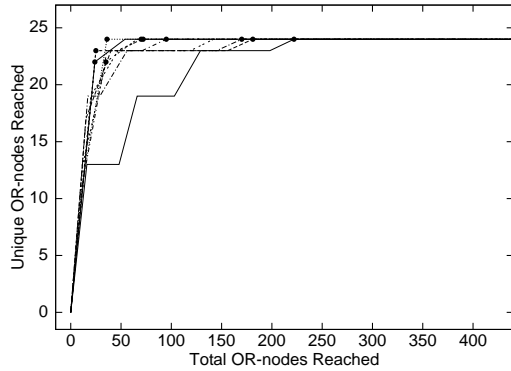
Our approach is as follows:

- Start with a sample of models;
- Infer from the sample *sanity constraints*, i.e., repeated features of those models;
- Build a model generator consistent with those sanity constraints.
- Define an executable measure of testability.
- For a large number of models from the generator, apply the testability criteria.
- Search the results for key factors that most improve testability.
- Impose those key factors on the model generator and generate more models.
- Return key factors for generating models that score higher on the testability measure.

An important precondition for such a study is that a sufficiently large log of testability results can be generated. We focus here on *partial random search* of *finite-state machines* (FSMs) since we have recently shown [13] that such a search is surprisingly effective, scalable, and rapid enough to quickly build a very large log of testability results. For example, Figure 1 (from Menzies et.al. [13]) shows ten random search trials for 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. Our random search algorithm is very simple, yet can handle state spaces much larger than model checkers. For example, Figure 2 shows random search results for a very large FSM model. The composite FSM representing all interleavings of the individual machines in the Figure 2 model would require at most 2.65×10^{178} states (model checkers' state of the art is about 10^{120} [3]).

The methodology of this paper requires a commitment to the kind of models being explored and an executable definition of testability. This paper explores software testability in

¹Submitted to the 17th IEEE International Conference on Automated Software Engineering, September 23-27, 2002, Edinburgh, UK <http://ase.cs.ucl.ac.uk/>. Date: May 12, 2002. WP: ase.tex.



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.

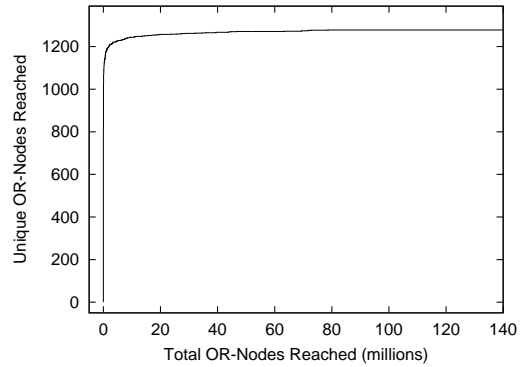
Figure 1. Random search of AND-OR graphs representing FSM models is effective in finding errors.

the context of *finite-state machines* (FSMs). Software systems with individual concurrent processes are often modelled as a *communicating* FSMs. FSMs are very simple structures to define, so we can easily implement a semi-random generator and quickly generate a large number of FSMs. We say *semi-random* since our FSM generator contains certain sanity constraints that block the generation of bizarre FSMs.

Figure 3 illustrates our executable definition of testability. Consider three possible results for a search over FSM models. Given some input, if the number of unique outputs (nodes in the AND-OR graph representing modelled program behavior) found as a result of that input rises quickly to a level plateau, a small number of tests will likely find everything it would be possible to find with many more. We say that such an FSM represents a program that is *easy to test*. A quicker rise to a higher plateau indicates that the program is *easier to test*. But if the search never reaches a plateau we infer that even after many tests more tests might give us more information, so the program is very *difficult to test*.

Generating large logs of testability results is pointless unless some tool exists for condensing those logs. We use the *TAR2 treatment learner* [12] to summarize our testability results for random search over FSMs. For example, we try to find out which is more testable: a model with many small finite-state machines or a model with a few large finite-state machines.

The next section contrasts standard notions of testability with the specific definition of testability explored here. This is followed by a formal definition of the type of FSM



Random search results for a very large randomly generated FSM model, for which the composite FSM would require at most 2.65×10^{178} states.

Figure 2. Random search of AND-OR graphs is scalable to very large models.

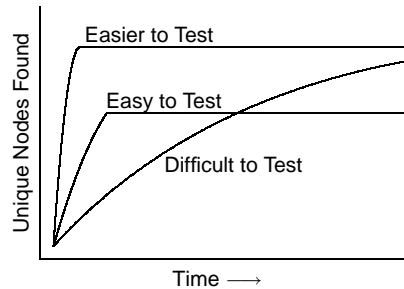


Figure 3. Intuitive testability interpretation of search results.

models used in our experiments; a description of how we translate these models into AND-OR graphs and test them by partial random search; and an introduction to the *TAR2* tool, which can be used to extract salient details from a large amount of data. Section 5 presents our experiments. We semi-randomly generate a large number of models and then use *TAR2* to determine what attributes make models more or less testable. To confirm *TAR2*'s conclusions, we generate a new set of models, which, as predicted by *TAR2*, are significantly more testable. We conclude by suggesting how *TAR2*'s conclusions might be applied to software design.

2 On Testability

In this paper we use the quick random search as a way of measuring the *testability* of FSM models. A model for which the search is able to more quickly find more information is considered more testable. Karoui et.al. have proposed a set of formally defined parameters influencing the testability of finite-state machine models [7, 8]: *controlla-*

bility, fuzziness, state-characterization degree, abstraction degree. Their approach provides a formula for determining each of these attributes. However, their definition is hard to operationalize on a large-scale since it requires a great deal of *white box* information about the finite-state machine implementation. The following fanciful analogy contrasts our work with that of Karoui et.al.: their method is like measuring the kinetic energy of a billion molecules and adding it all up in order to determine the temperature of a liquid. Our approach is to just use a simple thermometer capable of quickly comparing the temperature of one liquid to another.

The definition of testability used in this paper is unique. However, we claim that it 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 [14], 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 testability can be estimated (and this is not an easy task), it can serve as a basis for inferences on system verification based on test results. According to Hamlet and Voas [5], repeated failure-free executions of a program with a high probability of revealing failures (if they exist), should provide higher confidence in assessed reliability. Bertolino and Stringini dispute this point of view [2], because if faults remain in the testable program, there is a higher probability they will cause a failure in field use. Therefore, the reliability of a testable program does not have to be higher than the reliability of an untestable program, given that the two passed the same number of tests. Testable software designs should facilitate *revealing the faults* during verification process, without creating a greater probability of failure in operation.

Our definitions of *easy* and *easier to test* FSMs follow from the above mentioned conclusions. If (1) a random search procedure over a NAYO graph reveals many unique reachable nodes in the model quickly, and if (2) 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 will be difficult to reach in the operational environment too; hence the operational failure probability, due to testable design of the model, does not increase. Our experiments aim at answering the critical aspect of software testability, i.e., which attributes of the FSMs make the model more testable.

3 Finite State Machines

This section begins with our formal definition of *communicating finite-state machines* (3.1), or FSMs. This material is condensed from [13].

Model checking tools use the same form, with some minor variations, to represent programs with concurrent processes [6]. To verify that a model matches a property specification, a model checker must build an exponentially large *composite* finite-state machine representing all possible interleavings of the individual FSMs in the original model. We show how a type of AND-OR graph may be used to represent the same information, with size just polynomial in the size of the input (3.2). We then describe the partial random search procedure used to search our AND-OR graphs (3.3).

3.1 Communicating Finite-State Machines

We define a system S of communicating FSMs in the following way:

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

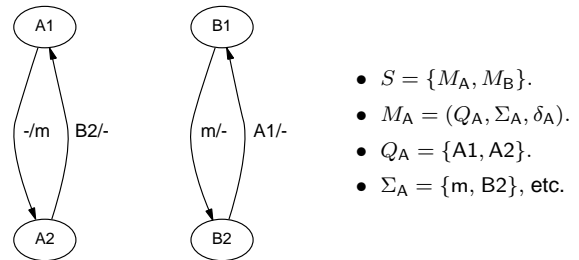


Figure 4. A system of communicating FSMs (m is a message passed between the machines).

Figure 4 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*. We have observed that in the variety of existing FSM schemes there are two different ways individual machines communicate. Because of this, we define two different kinds *input/output symbols* (included in Σ):

1. A transition in one machine may be triggered by the fact that another machine is in a particular state, or the

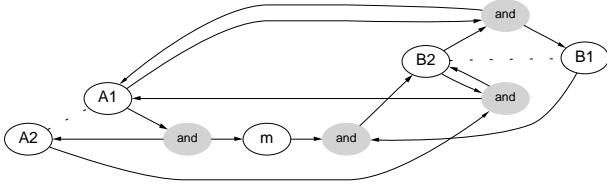


Figure 5. NAYO graph equivalent to FSM shown in Figure 4 (NO-edges dotted, AND-nodes shaded).

effect of a transition may be to change the state of another machine.

2. A transition may be triggered by a *message* received from another machine, or the effect of a transition may be to send a message.

The key difference between states and messages is in their use as transition inputs. 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.

3.2 NAYO Graph Translation

Figure 5 shows an AND-OR graph equivalent to the communicating FSM model shown in Figure 4. We call this type of AND-OR graph a NAYO [11] since it contains:

- A set N of undirected NO-edges connecting incompatible nodes.
- A set A of AND-nodes—an AND-node is TRUE if all of its YES-edge parents are TRUE.
- A set Y of directed YES-edges.
- A set O of OR-nodes—an OR-node is TRUE if any of its YES-edge parents are TRUE.

A close look at Figure 5 reveals something strange: there is an edge from node $A1$ to the upper right AND-node, and another edge going from the AND-node back to node $A1$ (the same thing occurs with node $B2$ and the lower right AND-node). To understand this, we reiterate the point made above, that there are two different ways FSMs communicate: *states* and *messages*. And the key difference between them is that messages are *consumed*. We use a simple trick to represent this in a NAYO graph. First, we define the search so that any time an AND-node is reached its parents are consumed—they are no longer available to be used again. Then, for nodes representing the *state-from-another-machine* type of input that should not be consumed (e.g., $A1$), we add an extra edge from the AND-node back to the

state node—we consume but then immediately regenerate the state node, so that it is available to be used again.

In general, for a system of k FSMs with n states and m single-input, single-output transitions per machine, the resulting NAYO has:

- mk AND-nodes + nk OR-nodes = $O((m + n)k)$ nodes.
- $4mk$ YES-edges + $(n/2)(n - 1)k$ NO-edges = $O((m + n^2)k)$ edges.

A FSM composite (the type of graph constructed by a model checker to represent a system of communicating FSMs) for the same system will in the worst case require $O(n^k)$ states and $O(n^{k-2})$ transitions [6].

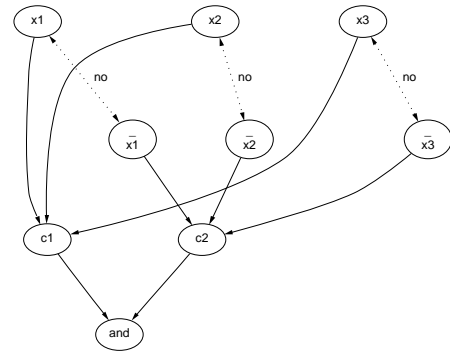


Figure 6. NAYO graph representing the 3SAT query $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$.

Unfortunately, the problem of determining whether a particular node in the NAYO graph can be reached is NP-complete,¹ which we show here in two steps.

$3SAT \leq_P$ NAYO search (NAYO search is at least as hard as the 3SAT problem, which is known to be NP-complete): for the 3SAT problem we have a Boolean expression that is the conjunction of a series of clauses, each of which is the disjunction of 3 literals. A literal is either a variable (x_i , for example) or its negation (\bar{x}_i). The problem is to determine whether the expression is *satisfiable*; that is, does there exist an assignment of values to the variables that satisfies the total conjunction? Figure 6 shows a NAYO graph representing a very simple 3SAT query. A NAYO graph for a 3SAT query will have a single AND-node; if this AND-node can be reached then the original 3SAT query is satisfiable.

NAYO Search \in NP: clearly we can verify a NAYO search solution in polynomial time; we would (1) verify that the solution is a valid path of YES-edges, which requires

¹NP is the class of problems for which a solution can be verified in polynomial time (the time required is a polynomial function of the input size); an NP-complete problem is (1) at least as hard as all problems in the class NP and is (2) itself in NP.

$O(n-1)$ time (where n is the number of nodes in the NAYO graph); (2) verify that no two nodes in the solution path are connected by a NO-edge, which requires $O(n(n-1))$ time.

3.3 NAYO Random Search

Our NAYO random search is designed to solve the following problem: given some (not necessarily consistent) input set of OR-nodes, find an output set consistent with at least part of the input, and make that output set as large as possible. Ideally the output set contains an OR-node for a state in each of the finite-state machines from the original model—if so, the output is equivalent to one of the states in the composite finite-state machine that would be searched by a model checker (and we have found it without explicitly constructing the composite). But in general, because the random search is not exhaustive, it may not tell us quite as much as a more time- (and space-) consuming technique; that is, the output set will constitute a *partial* description of a state in the composite.

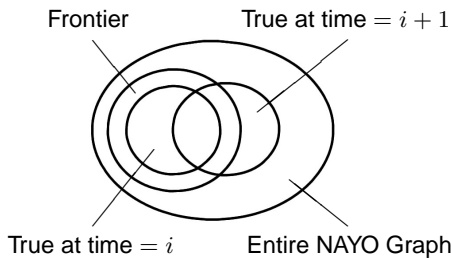


Figure 7. Sets involved in successive iterations of the random search procedure. shown in Figure 8.

Once we have the first output set, we use it as the input set for time $i + 1$. Figure 7 shows the sets of nodes involved in successive iterations of the random search procedure. At $time = i$, the search builds an output set of consistent nodes. These are marked *true at time = i* in Figure 7.

The set marked *frontier* is the set of nodes that will serve as input for the next iteration. The frontier includes (1) all nodes true now and (2) nodes that are *almost* true, i.e., nodes that are implied by but contradict nodes true now. In the next iteration ($time = i + 1$) we start with the frontier (which will include contradictions) as the input set and use it to build a consistent output set of nodes true at $time = i + 1$.

Figure 8 shows the random search procedure used to explore NAYO graphs. Each time the search comes to a node its *wait* field is decremented. When $wait = 0$, the node is *reached*. An OR-node's *wait* need only be decremented once, because we only need to reach it via one of its parents; so OR-nodes *wait* fields are initialized to 1 (line 1). To

```

1: OR-nodes' wait field ← 1.
2: AND-nodes' wait field ← |parents|.
3: while (time ≤ MAX) do
4:   while (Q ≠ ∅) do
5:     n ← pop(Q).
6:     if (n not disqualified) then
7:       Mark n true at current time.
8:       for (∀ n' linked to n by a NO-edge ) do
9:         Mark n' disqualified at current time.
10:      end for
11:      for (∀ YES-edge children n' of n) do
12:        Decrement n' wait field.
13:        if (n' wait = 0) then
14:          Mark n' reached at current time.
15:          Q ← n' at random index.
16:        end if
17:      end for
18:    end if
19:  end while
20:  Q ← all nodes reached at current time at random index
    (including nodes disqualified at current time).
21:  Reset all other nodes' wait fields (as in lines 1-2).
22:  Increment time.
23: end while

```

Figure 8. Random search procedure for NAYO graphs.

reach an AND-node, we must first reach all of its parents, so its *wait* field is initialized to its number of parents (line 2).

The central part of the search procedure occurs in lines 4-19. We begin with an input set of nodes in the Q , in no particular order. The first node is removed from the Q (line 5). If it has not been disqualified, i.e., it does not contradict some node we already believe true at the current time, we explore its children. All children via NO-edges are disqualified (line 9). The *wait* fields of all children via YES-edges are decremented (line 12), and if any are decremented all the way to zero, they are put into the Q at some random index (line 15). This process continues until the Q is empty (line 4).

Once all nodes in the Q have been processed, lines 20-22 set us up for the next iteration. At this point there is a set of nodes marked *true* at the current time, which is a subset of the nodes marked *reached* at the current time (some nodes are reached but disqualified, so they are never marked *true*). The *true* set corresponds to the set *true at time = i* in Figure 7, and the *reached* set corresponds to the *frontier* set in Figure 7. The *reached* set is put back into the Q (line 20) to serve as input for the next iteration. All other nodes' *wait* fields are reset, and the time is incremented (lines 21-22).

Figure 9 shows a comparison of search results for (1) exhaustive search of a composite finite-state machine, which generates an execution path of fully defined global states, and (2) our NAYO search, which generates an execution path of partially defined global states (without explicitly constructing the exponentially large composite FSM representing the global system). The search continues for a spec-

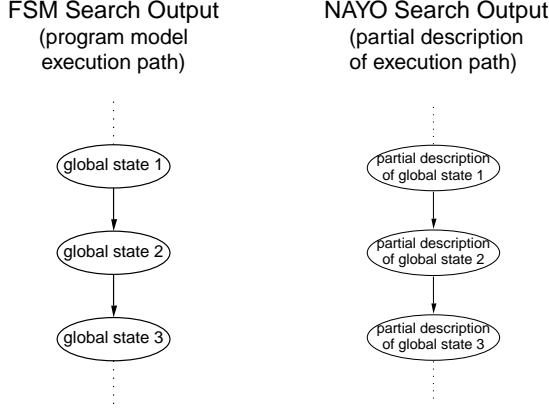


Figure 9. Comparison of output type for exhaustive FSM search and our partial random NAYO graph search.

ified number of iterations or until it hits a dead end. The search is random in that, when there are two or more contradictory nodes that might be added to the output set, the choice of which node to add is random.

4 The TAR2 Treatment Learner

The previous section shows how to *generate* data from random search over AND-OR graphs. This section discusses a method for *summarizing* that generated data using the TAR2 *treatment learner*. Most of this material comes from [12].

Unlike standard machine learners, TAR2 does not learn *descriptions* of classes. Rather, it learns the *differences* between classes. TAR2 assumes that classes are ordered by *score* (a domain-specific measure); classes with a high score are considered better than classes with a low score, and the most desirable class (which has the highest score) is called the *best* class. TAR2 finds rules that predict both an increase in the frequency of better-class cases and a decrease in the frequency of cases in worse classes; that is, TAR2 finds rules that drive cases toward the best class and away from the worst.

TAR2 outputs *treatments* rather than classifications; a treatment is an attribute range (or a conjunction of attribute ranges) that can be used as a constraint on future input cases—a guide for creating an input set of cases that fall into better classes. For example, Figure 10 shows a small training set with four attributes (outlook, temperature, humidity, wind) and three classes (none, some, lots). TAR2 assigns a score to each class, for example, the worst class, *none* (the amount of golf played = none), might be scored 2, the class *some* scored 4, and the best class, *lots*, scored 8. TAR2 looks for the attribute ranges in which cases more

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

Figure 10. A simple training set for TAR2 (a log of golf-playing behavior).

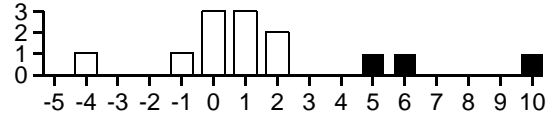


Figure 11. Δ distribution for golf data shown in Figure 10 (very high Δ values are shown in black; the y -axis is the number of attribute ranges with a particular Δ).

frequently fall into higher-score classes (and fall into lower-score classes less frequently).

Let $a = r$ be some attribute range, e.g., *outlook = overcast*, and let $X(a = r)$ be the number of times that attribute range occurs in class X , e.g. $lots(outlook = overcast) = 4$. $\Delta_{a=r}$ is a measure of the significance of $a = r$ in improving the average class of cases restricted to that attribute range. $\Delta_{a=r}$ uses the following definitions:

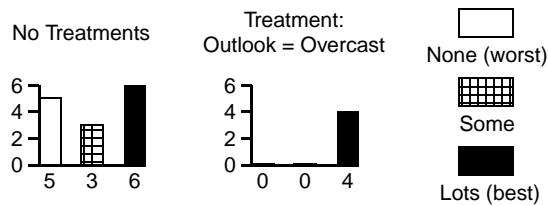
- *best*: the class with the highest score, e.g., *lots*.
- *rest*: all other classes, e.g., $\{none, some\}$.
- *score*: the score of a class X is $\$X$.
- $|cases(a = r)|$: the number of cases in which $a = r$ occurs.

The $\Delta_{a=r}$ value is calculated as:

$$\sum_{X \in rest} \left(\frac{(\$best - \$X) \cdot (best(a = r) - X(a = r))}{|cases(a = r)|} \right)$$

The attribute ranges in our golf example generate the Δ histogram shown in Figure 11. A *treatment* is a subset of attribute ranges with outstanding $\Delta_{a=r}$ values. For our golf example, such attributes can be seen in Figure 11: they are the outliers with very large Δ values on the right.

To *apply* a treatment, TAR2 rejects all cases not included in the attribute range (or contradicting the conjunction of



With no treatments, we play *lots* of golf only $\frac{6}{5+3+6} = 57\%$ of cases; with a treatment restricting the attribute *outlook* to *overcast* only, we play *lots* of golf in 100% of cases.

Figure 12. A treatment to maximize the amount of golf played.

attribute ranges) that makes up the treatment. The ratio of class scores for the remaining cases is compared to the ratio of class scores for all cases in the original input set. The *best treatment* is the treatment that most increases the relative percentage of preferred classes, e.g., *outlook = overcast*. Figure 12 shows the class distribution before and after applying that (best) treatment (if we pick a holiday location where the weather forecast is overcast, we can then expect to play *lots* of golf in 100% of cases).

5 Testability Experiments

In this section, we describe our method for randomly generating program models with attributes typical of FSMs (5.1). We then use the TAR2 tool described in the previous section to explore search data from 15,000 semi-randomly generated FSM models. TAR2 gives us some predictions about what FSM model attributes make a model easiest to test by random search (5.2).

5.1 Semi-Random Generation of Program Models

We use the attributes listed below to describe the generation of semi-random finite-state models.

1. The number of individual finite-state machines in the system.
2. The number of states per finite-state machine.
3. The number of transitions per machine.
4. The number of inputs per transition that are states in other machines.
5. The number of unique *consumable* messages that can be passed between machines.
6. The number of inputs per transition that are consumable messages.
7. The number of outputs per transition that are consumable messages.

To illustrate these attributes, consider the simple example model shown in Figure 4. That model has two finite-state machines, with two states per machine and two transitions per machine; there are two transition inputs that are states from another machine (A_2, B_2) and one unique *consumable* message (m), which is an input for one transition and an output for another.

It may seem strange that there is no “maximum number of outputs per transition that are states ...” This is because the relationship between finite-state machines expressed by a transition with such an output is equivalent to the relationship expressed by an input that is a state in another machine. This is not true for message inputs and outputs, which must be generated by one transition in order to be consumed by another.

Our model generator is semi-random because it imposes certain *sanity constraints* on the generation process. These sanity constraints are designed to keep the random models similar to real models. First, all machines have at least 2 mutually exclusive states (it wouldn’t make sense to define a machine with 0 or only 1 state, although it would be possible to represent it in the NAYO graph). For transitions, the following rules were applied:

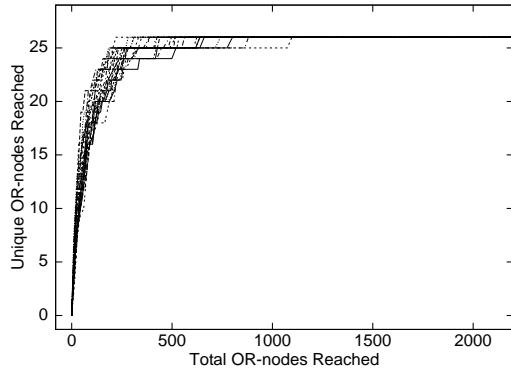
- The *current state* and *next state* must come from the machine in which the transition is defined and must not match.
- Inputs that are states must come from *other* machines, and none may be mutually exclusive (the transition could never occur if it required mutually exclusive inputs).
- The set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

Figure 13 shows search results for a semi-randomly generated model with attributes similar to real models discussed in the introduction (see Figure 1). As with the real models, here we see a quick rise to a level plateau in all trials of the experiment.

5.2 Finite-State Model Attributes’ Influence on Testability

To explore the relationship between testability and the finite-state model attributes listed in section 5.1, we generated 15,000 NAYO graphs with the following attribute ranges:

1. 2–20 individual FSMs.
2. 4–486 states (states within machines—the composite FSM representing the entire system would exponentially many more states).



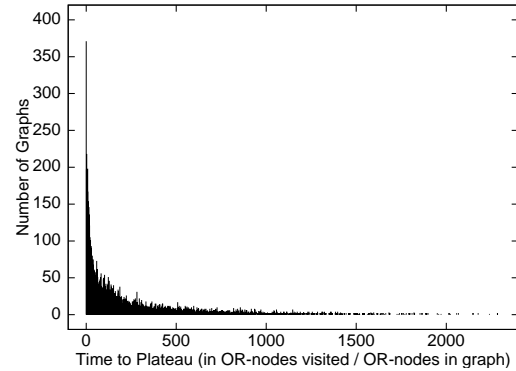
A semi-random model with attributes like those of Dekker (several small FSMs, transitions triggered by states from other machines).

Figure 13. Search results for semi-randomly generated FSM model.

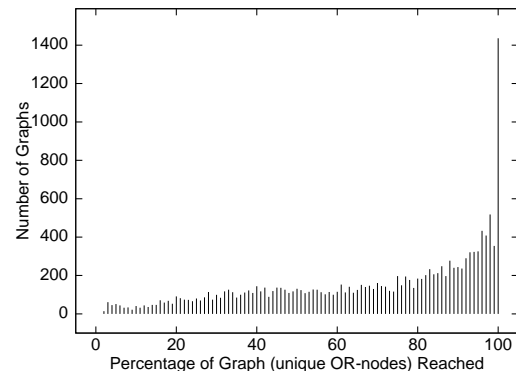
3. 0–272 transitions (the composite FSM would have many more transitions).
4. 0–737 transition inputs that are states in other machines.
5. 0–20 unique consumable messages.
6. 0–647 transition inputs that are consumable messages.
7. 0–719 transition outputs that are consumable messages.

Figure 14 shows a summary of search results for semi-randomly generated models. The top histogram summarizes *time-to-plateau* results, e.g., *time-to-plateau* for approximately 375 models was $0 (< 1 \times \text{graph size})$. The average value was about $208 \times \text{graph size}$. 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 14 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 14 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 know 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 14?



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



Average plateau height = 69.39%.

Figure 14. Summary of time-to-plateau (top) and plateau height (bottom) results for 15,000 models.

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 15, 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 250 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 16 shows a summary of re-

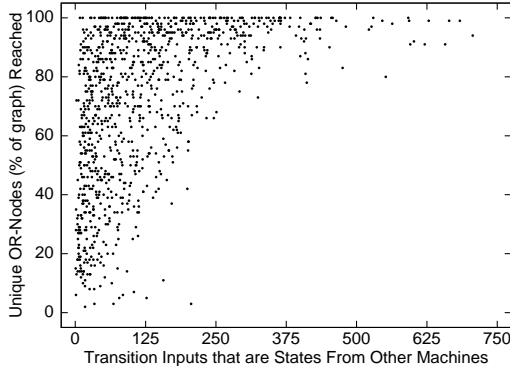


Figure 15. The number of transition inputs that are states from other machines vs. plateau height.

sults from a series of experiments, in which we tried to determine which combinations 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 16 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 17 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.

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 17 shows this paper’s most significant result. It is possible to learn parameters of an FSM that significantly

	← 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		(not significant)	
Message Inputs		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		(not significant)	
Message Inputs		lowest (0–129)	
Message Outputs	lowest (0–143)		

Figure 16. Best and worst treatments learned by TAR2.

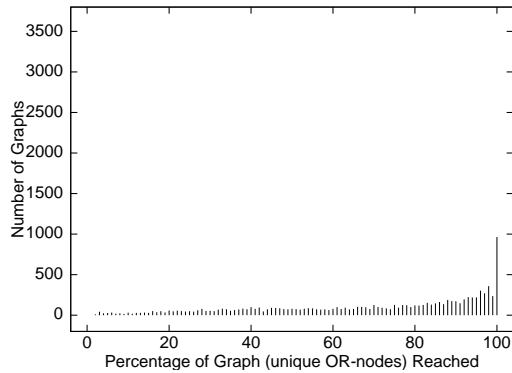
improve FSM testability. In this case the improvement was a change in the average plateau height from 69% to 91%.

6 Conclusion

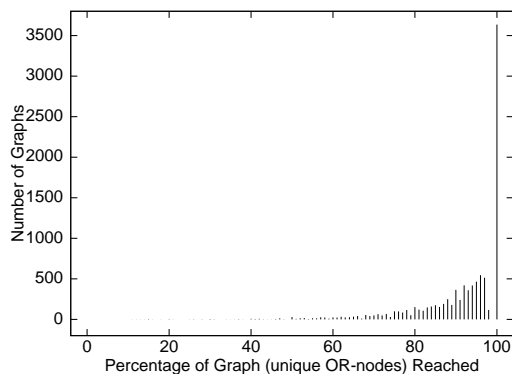
We have proposed a general method for assessing the testability of a certain class of models, given a specific executable definition of testability. We have instantiated that general method in the context of *plateaus* in the rate of new nodes found by *random search* over *FSMs*. We have said that an FSM is *more* testable if it plateaus at a higher value. Using TAR2, we have found FSM parameters that result in such higher plateaus.

Curiously, our experiments indicate that smaller FSMs are not necessarily more testable. Larger, more complex FSMs are likely to fall in the middle-to-high testability range, and small, simple FSMs are likely to be either very testable or very difficult to test. It is connectedness (the number of transition inputs and outputs), not size, that is most important for testability.

Also, we found transition inputs that are states from other machines to be more significant for testability than inputs that are messages passed between machines. A designer may often have a choice of whether information should be available globally (e.g., states in one machine vis-



Original search data (i.e. Figure 14 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%.

Figure 17. Comparison of plateau height for original search data (top) and new data based on TAR2's suggested treatments.

ible to another) or should be passed to a specific destination and hidden from others (e.g., messages). Usually global variables are thought of as a design liability; the scope of information is to be limited as much as possible in order to limit the propagation of errors. But our experiments show there is actually a tradeoff here: the designer may need to choose whether to make errors easy to find (testability) or less catastrophic when they do occur. The *exception handling* capability of some languages addresses this principle to some extent, but can only deal with a set of anticipated errors—there will always be strange problems not caught.

We suggest the following future work:

- Apply the TAR2 treatment learning technique to a wide range of real models in order to confirm our conclusions about which attributes are most significant;
- Develop efficient ways to manipulate models, changing their attributes to make them more testable while

changing the implementation they represent as little as possible;

- Instantiate our model testability framework for other model types and for other definitions of testability.

References

- [1] IEEE glossary of software engineering terminology, ANSI/IEEE standard 610.12, 1990.
- [2] A. Bertolino and L. Strigini. On the Use of Ttestability Merasures for Dexpendability Aassessment. *IEEE Transactions on Software Engineering*, 20(2):97–108, February 1996.
- [3] E. Clarke, O. Grumberg, and D. 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] D. Hamlet and J. Voas. Faults on its sleeve: Amplifying software reliability testing. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis, Cambridge, MA*, pages 89–98, June 1993.
- [6] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990. Available at <http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html>.
- [7] K. Karoui and R. Dssouli. Ttestability Analysis of the Communication Protocols Modeled by Relations, 1996. Available at <http://citeseer.nj.nec.com/147902.html>.
- [8] K. Karoui, A. Ghedamsi, and R. Dssouli. Study of Some Influencing Factors in Ttestability and Diagnostics Based on FSMs, 1996. Available at <http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html>.
- [9] M. Lowry, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, ASE'98: Automated Software Engineering*, pages 322–331, 1998.
- [10] T. Menzies and B. Cukic. How many tests are enough? In S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering, Volume II*, 2002. Available from <http://tim.menzies.com/pdf/00ntests.pdf>.
- [11] T. Menzies, B. Cukic, H. Singh, and J. Powell. Testing Non-determinate Systems. In *ISSRE 2000*, 2000. Available at <http://tim.menzies.com/pdf/00issre.pdf>.
- [12] T. Menzies and Y. Hu. Constraining Discussions in Requirements Engineering via Models. In *First International Workshop on Model-Based Requirements Engineering*, San Diego, CA, 2001.
- [13] T. Menzies, D. Owen, and B. Cukic. Saturation Effects in Testing of Formal Models. In *Submitted to ISSRE 2002*, 2002. Available from <http://tim.menzies.com/pdf/02sat.pdf>.
- [14] J. Voas and K. Miller. Software testability: The new verification. *IEEE Software*, pages 17–28, May 1995. Available from <http://www.digital.com/papers/download/ieeesoftware95.ps>.