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
{dowen|cukic}@csee.wvu.edu, tim@menzies.com

1 Introduction

How should we test software? Given a range of possible test methods, when is one technique preferred to another?

Typically, these kinds of questions are answered with reference to the inherent properties of the assessment mechanism. For example, Lowry et.al. [7] and Menzies & Cukic [8] contrast the costs and defect detection decay rates of formal methods, white box testing, and black box testing. That analysis made assumptions about the completeness of the search and the cost of setting up each run. A drawback with that kind of analysis is it is silent about the model being searched for defects[1].

This paper studies how details of a particular model can effect the efficacy of a search for detects. We find that if the test method is fixed, we can identity classes of software that are more or less testable. Using a combination of *model mutators* and *machine learning*, we find that we can isolate topological features that significantly change the effectiveness of a defect detection tool. More specifically, we show that for one defect detection tool (a stochastic search engine) applied to a certain representation (finite state machines), we can increase the average odds of finding a defect from 69% to 91%. The method used to change those odds is quite general and should apply to other defect detection tools being applied to other representations.

These results draw into question the results like those of Lowry et.al. and Menzies & Cukic. If simple changes to a model's topology can increase detect detection to near 100%, then the efficacy of a defect detection tool must be assessed *in conjunction with* the program being assessed.

[0]17th IEEE International Conference on Automated Software Engineering, September 23-27, 2002, Edinburgh, UK http://ase.cs.ucl.ac.uk/. Date July 12, 2002. WP: ase.tex.

[1]Exception: It is widely acknowledged that model checking is exponential on model size and so only a small part critical region of a total system should be assessed via model checking.

2 Experimental Design

This study used the following tools: some finite state machines (FSMs); the LURCH1 *stochastic search* engine [10]; a model mutator; and the TAR2 *treatment learner* [9]. This section describes those tools.

2.1 FSMs

An FSM has the following features:

- 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.
- $\delta : Q \times B \longrightarrow Q \times B$, where B is a set of zero or more symbols from Σ, is the transition function.

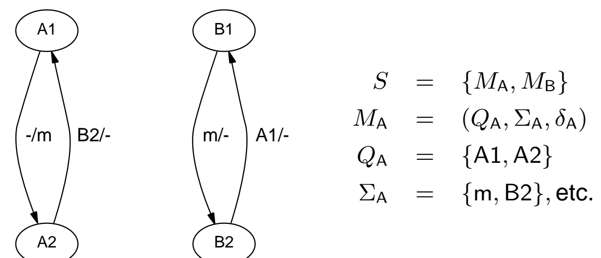


Figure 1. A system of communicating FSMs ("m" is a message passed between the machines).

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.

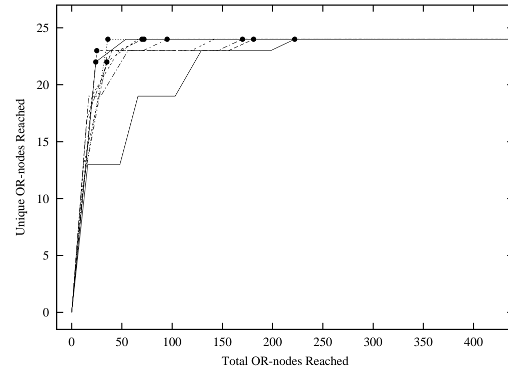
2.2 LURCH1

A complete search for potential violations of temporal properties within an FSM is intractable. The model checking community has tried to optimize this search for decades. While various techniques reduce the search space in certain limited domains, the general problem remains intractable (see the review in [5]).

When a complete search is too slow, *stochastic search* is an alternative that can be surprisingly effective. A repeated result from the artificial intelligence literature, is that randomly selected search pathways can find nearly optimal or optimal results for large problems [4, 6, 11]. These results prompted our development of LURCH1, a random search engine for formal specifications [10].

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 stochastic nature arises from how the search proceeds after a new assertion is made. If all the preconditions 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 and the node x will be blocked.



Random search results for model of Dekker's solution to the two-process mutual exclusion problem (the model comes from Holzmann [3]). Dots show when an error added to the model is found by the search. The error is found in every case.

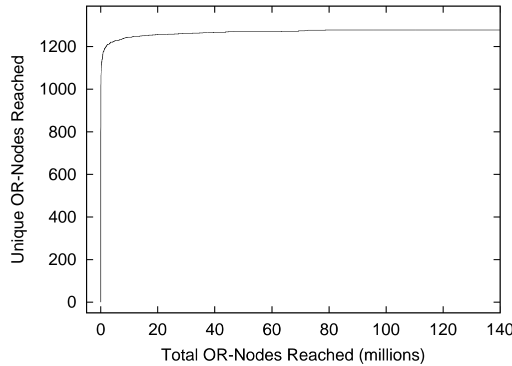
Figure 2. Random search of AND-OR graphs representing FSM models is effective in finding errors.

The stochastic search of LURCH1 is theoretically incomplete but, in practice, it is surprisingly effective. For example, Figure 2 (from Menzies et.al. [10]) 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 [3]). 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 2.65×10^{178} states. This is well beyond the capability of model checking technology (10^{120} states according to [2]).

2.2.1 LURCH1 and 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 [13], testability is defined as "the degree to which a system of components facilitates the establishment



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

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

of test criteria and the performance of tests to determine whether those criteria have been met". Voas and Miller [12], and later Bertolino and Stringini [1] 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[2].

2.3 Model Mutator

LURCH1 was run over 15,000 FSMs generated semi-randomly. 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 [10]).

[2]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.

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.

2.4 Treatment Learning

The results of the 15,000 were analyzed by the TAR2 *treatment learner* [9]. Unlike standard machine learners, TAR2 does not learn *descriptions* of the different classes in the training set. Such a description can be very large. A smaller description of the essential features of a training set is found by TAR2 and contains 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.

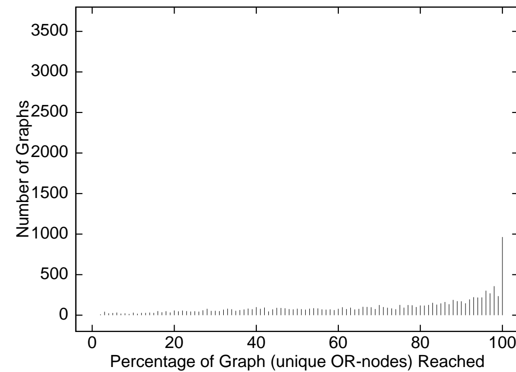
For this application, TAR2's class scores reflected the plateau height of LURCH1.

3 Results

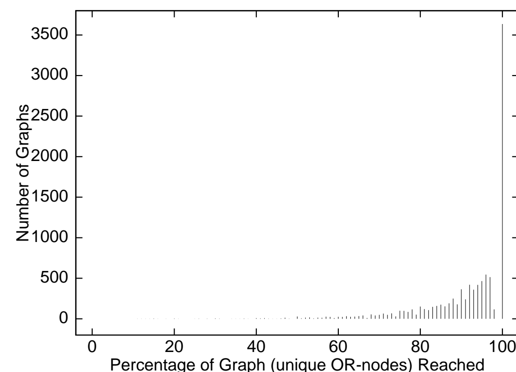
Figure 4 shows the results of applying TAR2 to the 15,000 runs of LURCH1 over the semi-randomly generated FSMs. The bottom half of Figure 4 shows which attributes have the greatest affect on testability, given that the top three are held low. The most significant attribute is *state inputs,* followed by *message inputs* and *message outputs.* To verify this result from TAR2, we generated 10,000 more FSMs according to our sanity rules, but with the added constraints of the bottom half of Figure 4. Figure 5 shows a comparison of plateau height (our indicator of testability) for the original data (top) and the new 10,000 runs (below). Note that TAR2 has learnt FSM parameters significantly improve FSM testability. In this case the improvement was a change in the average plateau height from 69% to 91%.

	⟵ 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)

Figure 4. Best and worst treatments learned by TAR2.



Original search data— average plateau height = 69.39%.



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

Figure 5. Comparison of plateau height for original search data (top) and new data based on TAR2's suggested treatments.

4 Conclusion

Simple changes to a model's topology can increase detect detection to near 100%. These changes can be learnt automatically using model mutators (constrained by sanity checks) and treatment learning. We recommend this method of assessing the efficacy of a defect detection tool *in conjunction with* the program being assessed.

References

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

[2] E. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[3] Gerard J. Holzmann. Basic SPIN Manual. Available at `http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.htm`.

[4] H. Hoos and C. Boutilier. Solving combinatorial auctions using stochastic local search. In *Proc. of AAAI-2000*, pages 22–29. MIT Press, 2000.

[5] M. Houle, T. Menzies, and J. Powell. A fast search for temporal properties of requirements, 2002. Available from `http://tim.menzies.com/pdf/02sp2.pdf`.

[6] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, Aug. 4–8 1996. AAAI Press / MIT Press. Available from `http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps`.

[7] 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.

[8] 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`.

[9] T. Menzies and Y. Hu. Constraining discussions in requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://tim.menzies.com/pdf/01lesstalk.pdf`.

[10] T. Menzies, D. Owen, and B. Cukic. Saturation effects in testing of formal models. In *ISSRE 2002*, 2002. Available from `http://tim.menzies.com/pdf/02sat.pdf`.

[11] B. Smith and M. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):155–181, 1996.

[12] 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`.

[13] IEEE glossary of software engineering terminology, ANSI/IEEE standard 610.12, 1990.