Saturation Effects in Testing of Formal Models

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

Formal analysis of software is a powerful analysis tool, but can be too costly. Random search of formal models can reduce that cost, but is theoretically incomplete. However, random search of finite-state machines exhibits an early saturation effect*, i.e., random search quickly yields all that can be found, even after a much longer search. Hence, we avoid the theoretical problem of incompleteness,* provided *that testing continues until after the saturation point. Such a random search is rapid, consumes little memory, is simple to implement, and can handle very large formal models (in one experiment shown here, over* 10^{178} *states).*

1 Introduction

Formal modelling, analysis and verification are very active research areas in software assurance. Proponents of formal methods promise improved software reliability (see the long list of applications in [26]). However, doubts exist related to the practicality and cost of applying formal methods:

- The cost of developing the formal model or a query, referred to in the rest of this paper as the *writing cost*. Often PhD-level mathematical expertise is required for writing such models.
- The cost of executing queries of the formal model, i.e., the *running cost*. A formal query can be impractically slow to execute since it should explore all possible interactions. Such an exploration may require exponential CPU or memory resources.
- The cost of introducing changes into the formal model, i.e., the *rewrite cost*. Analysts often rewrite formal models into a more abstract and succinct form in an attempt to reduce the *running cost*.

Many researchers have tried to reduce these costs using a variety of methods, such as restricted modelling languages and temporal logic patterns discussed in the related work section. In summary, much progress has been made in reducing the *writing cost*, but the general problem of a high *running cost* persists despite decades of work.

In an attempt to minimize the *running cost* and thereby decrease the *rewrite cost* (which is associated with modifying models to decrease verification run times), we have been exploring formal models compiled into a variant of AND-OR graphs called *NAYO graphs* [36, 38, 39]. These NAYO graphs have the advantage that they can be auto-generated from commonly used representations, such as finite state machines, but searching them does not require exponential memory, as is required by standard model checkers working on finite-state models.

While NAYO graphs avoid the memory problems of model checkers, they may be very slow to search. As we shall see later on, a *complete search* of NAYO graphs is prohibitively long (exponential on the size of the graph), incurring unacceptable *running cost*.

The alternative to complete search is *incomplete search*. A repeated and recent result in the artificial intelligence literature is that such incomplete searches may be surprisingly effective. The logical form of complete search of a NAYO graph is similar to the SAT (satisfiability) problem [34] (in fact, exhaustive search of a NAYO graph is NP-complete [45]). *Random search* is an incomplete strategy often used for SAT problems:

- When competing constraints block progress, one constraint (selected at random) is favored.
- Future impasses are also anticipated and resolved randomly, i.e., this search finds a randomly selected subset of a formal model—this is what we mean by *incomplete*.
- Random search runs, resets, and retries N times. The best solution seen in any run is returned as the result.

A repeated result is that random search finds optimal or nearly optimal results for large satisfiability problems [28, 31, 51]. Further, random search finds a result even when

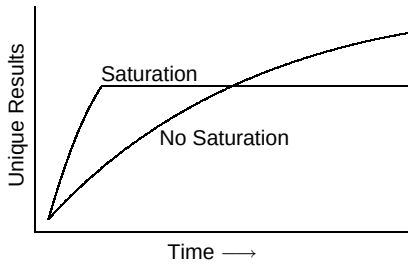


Figure 1. The saturation effect.

exhaustive search is not feasible. These results motivated us to explore the application of random search techniques in software engineering, investigating search over NAYO graphs generated from finite-state machines. Our research lead to the following discovery:

> Random search over formal models of software specifications exhibits a *saturation effect*.

An example of the saturation effect is shown in the *saturation* curve of Figure 1. In that curve, most of the states observed and analyzed by searching a (NAYO) graph are discovered very early in the process. Saturation usually manifests itself as a sharp initial increase in the number of unique states visited (states unobserved in previous searches), followed by a plateau where extra effort does not yield any more unique results.

Assessment methods that lack the saturation effect have the property that, the longer the assessment, the more unique results are found: see the *no saturation* curve of Figure 1. On the other hand, assessment methods that exhibit a saturation effect support early stopping rules. Such rules reduce the cost of formal analysis, i.e., the *running cost*. We can stop searching a formal model when it is very unlikely that prolonged search will uncover new results, i.e., once the saturation plateau is encountered.

Assessment methods using early stopping rules run the risk of false positives, i.e., reporting that no faults are present when further assessment would have found them. Hence, early stopping can only be endorsed for searches that exhibit *adequacy* and *flat plateau* properties:

Adequacy: An adequate assessment method does not fail to recognize faults in states visited (covered) prior to the occurrence of saturation.

Flat plateau: If the saturation curve results in a *flat plateau*, then additional errors, if present in the unseen portion of the formal model, are not detected. However, due to the flatness of the plateau, they are most likely to remain unnoticed upon system deployment, unless its operating regime undergoes drastic changes.

Our claim is that random search of NAYO graphs generated from finite-state machines demonstrates both adequacy and consistently flat plateaus. So it appears that early stopping is possible for testing of formal models, assuming we are using random search over NAYO graphs to test them. If true, this would eliminate a significant portion of the *running cost* and the *rewrite cost*.

The rest of this paper defends this claim. After a discussion on related work, we describe an environment for translating communicating finite-state machine software models into NAYO graphs. Multiple experiments with random search over these graphs demonstrate repeated occurrence of the saturation effect. Case studies will be presented suggesting that our method of testing formal models is *adequate* and generates *flat plateaus*.

2 Related Work

2.1 The Saturation Effect

Saturation is consistent with programs containing zones that are easily reachable and zones that are not reachable at all. Elsewhere, we have surveyed the numerous reports of this effect in the software engineering and knowledge engineering literature [36, 37]. This section is a brief sample of those surveys.

Horgan & Mathur [29] document systems where most program paths get exercised early with little further improvement as testing continues. Mutation testers often report that a small sample of their mutations[1] find as much as a much larger sample of mutations [2, 7, 42, 53].

Numerous researchers report that much of the apparent complexity of their program collapses into a very small zone through which all execution paths must traverse. For example, Avritzer et.al. [3] found that a sample of 6% of all inputs to a telecommunications expert system covered 99.9% of all inputs seen in one year's operation of that program. In other work, Bieman & Schultz [6] document a natural language processing system where less than 50 inputs were enough to exercise all du-pathways[2] in 95% of the system modules. A similar effect was noted by Harrold et.al. [22], who studied how control-flow diagrams grow as program size grows. A worst-case control-flow graph is one where every program statement links to every other statement, i.e. the number of edges in the graph grows with the square of the number of statements. However, for over 4000 Fortran routines and 3147 C functions, the control flow graph grows linearly with the number of statements. That is, at least in the systems seen in the Harrold et.al.

[1] A *mutant* of a program is a syntactically valid but randomly selected variation to a program; e.g. swapping all plus signs to a minus sign.

[2] A du-pathway is a link from where a variable is *defined* to where it is *used*.

study, the program pathways form single-parent trees. This collapsing effect has been observed in many domains: "collapsed" program portions have been called *master-variables* in scheduling [13], *backbones* in the satisfiability community [46, 50], and *minimal environments* in truth maintenance systems ATMS [15].

Saturation is such a widespread effect that it has prompted several mathematical studies. Menzies & Singh concluded that saturation is a property which should emerge in most programs (in the average case) [41]. Another simulation-based analysis, reported at ISSRE 2000 [38], made a similar conclusion.

2.2 Formal Modelling

In mission-critical or safety-critical applications, where required safety and reliability justify the high cost of software and system assurance, formal verification is often used—see the long list of applications in [26]. Running a single formal query can find many faults, if they exist. However, finding the same faults with specific test cases may require generating a large number of inputs and running them many times.

Another useful feature of formal analysis is that verification engines for formal models such as the SPIN *model checker* [26] return *counter examples* showing exactly how an invariant is violated. Such counter examples are useful in localizing and repairing faults.

Temporal logic is a useful notation for describing the temporal properties of a device. Temporal logic is classical logic augmented with temporal operators such as $\Box X$ (always X is true), $\Diamond X$ (eventually X is true), $\bigcirc X$ (X is true at the next time point), $X \mathcal{U} Y$: (X is true until Y is true). For example, consider the property

> Always, the elevator door never opens more than twice between the source floor and the destination floor.

Suppose P denotes elevator doors opening, Q denotes the arrival of an elevator at the source floor and R denotes the arrival at the destination floor. The following daunting expression models our rule about the elevator doors:

```
□((Q ∧ ◇ R) → ((¬ P ∧ ¬ R)
     𝒰 (R ∨ ((P ∧ ¬ R)
        𝒰 (R ∨ ((¬ P ∧ ¬ R)
           𝒰 (R ∨ ((P ∧ ¬ R)
              𝒰 (R ∨ (¬ P ∨ R)))))))))))
```

Automated model checking tools can search a formal representation of a problem (program) to find counter-examples to the supplied correctness constraint (e.g., the expression above). This search explores all the possible interactions within the program. In the worst case, the number of such interactions (system states) is exponential with respect to the number of different assignments to variables in the system. Hence, the *running cost* associated with a query can be excessive. This large *running cost* often forces analysts to shorten and rewrite the formal models or formal constraints. Such a rewrite incurs the *rewrite cost*. Also, in an attempt to simplify formal system representation, an analyst may unintentionally ignore some, potentially significant, system detail.

Significant research efforts target the reduction of writing and running costs for formal verification. For example, simplified modelling environments have been developed, such as SCR [23–25] or influence diagrams [40], through which users can express their models in simple and intuitive frameworks. Models written in these environments can be automatically mapped into model checkers, such as SPIN, thus combining easy modelling of requirements specifications with formal verification. Furthermore, Dwyer, Avrunin & Corbett [16, 17] identified *temporal logic patterns* within the constraints seen in many real-world properties models. For each pattern, they defined an expansion from the intuitive pseudo-English form of the pattern to a formal temporal logic formula. In this way, analysts are shielded from the complexity of formal logics. For example, our *writing cost* for the elevator expression was, in fact, minimal. We just looked up the "bounded existence" temporal logic pattern at `http://www.cis.ksu.edu/santos/spec-patterns/ltl.html` and extracted the expression associated with *transitions to P-states occur at most 2 times between Q and R*.

These and similar tools reduce the *writing cost* but do not necessarily reduce the *running cost* or the *rewrite cost*. The *rewrite cost* is incurred when the *running cost* is too high and the models or constraints must be abbreviated.

There is no guarantee that formal verification is tractable over the constraints and models built using temporal logic patterns and tools like SCR. Restricted modelling languages *may* generate models simple enough to perform rigorous formal verification, but the restrictions on the language can be excessive. For example, checking temporal properties within simple influence diagrams can take merely linear time [40], but such a language can't model common constructs such as sequences of actions or recursion. Hence, analysts may be forced to use more general modelling languages. Unfortunately, the high *running cost* of such general verification languages persists despite decades of work.

2.3 Reducing the Running Cost

Many researchers have explored reducing the *running cost* using a variety of techniques:

- *Symbolic model checking*: use binary-decision diagrams (BDD's) to succinctly represent the otherwise

excessively large state space [10]. Tools for working with BDD's are widely available—see (e.g.) bdd-portal.org.

- *Abstraction or partial ordering*: use only the part of the space required for a particular counter-example. Implementations exploiting this technique can constrain how the space is traversed [20] or constructed in the first place [49].
- *Clustering*: divide the system model into sub-systems which can be reasoned about separately [9].
- *Meta-knowledge*: study only succinct meta-knowledge of the space. One example used an eigenvector analysis of the long-term properties of the systems model under study [30].
- *Exploit symmetry*: find properties in some part of the system model, then re-use those counter-examples if ever those parts are found elsewhere in the model [8].
- *Semantic minimization*: replace the space with some smaller, equivalent space. For example, BANDERA [12] reduces both the system's writing cost and the running cost by automatically extracting (slicing) the minimum portions of a JAVA program's bytecodes relevant to the particular properties specified by the analyst.

While the above techniques have all been useful in some application domains, they may not be universally applicable. Certain optimizations require expensive preprocessing [30]. Also, these methods may rely on certain combinatorial features of the system being studied. Exploiting symmetry is only useful if the system under study is highly symmetric. Clustering generally fails for tightly connected models. Hence, in the general case, it seems that only relatively small models (although the maximum size is increasing due to increasing in processing power) can be formally verified using state-space search techniques.

2.4 Other Stopping Rules for Testing

Early stopping rules in testing have been thoroughly studied. Certification tests using reliability demonstration charts have been introduced by Musa, Iannino and Okumoto [43, 44]. A *reliability demonstration chart* is shown in Figure 2. There are three regions on the chart: *reject*, *test* and *accept*. A failure is plotted if it occurs during random testing, in which tests are selected according to the *operational profile* (an operational profile is a statement of what input values are expected at runtime). The vertical axis on the chart denotes the failure number, while the horizontal axis denotes normalized occurrence time (for example, occurrence time multiplied by the failure intensity objective). Depending on where the failure is plotted with respect to the graph regions, testing is stopped (with the program either accepted or rejected) or continued. The number of tests

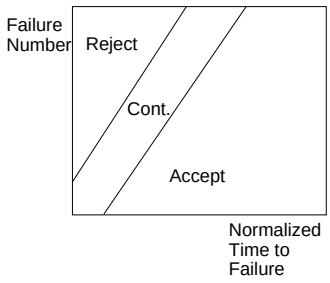


Figure 2. The reliability demonstration chart.

required for reliability certification test in this technique depends on the position of the lines between reject, continue and accept regions.

Another method for defining early stopping rules is to use Bayesian prior probabilities. Testing can stop early when priors stabilize and the probability of failure on demand is low. Consider the case when failure occurs within a run of 5,000 tests. Suppose that failure is detected at test 3,500. The problem is: how many additional tests need to be executed successfully following one or more failures, to be able to certify requested reliability without debugging? Littlewood and Wright [32] proposed one solution, based on Bayesian statistics. At the start of the certification test, compute n_1, the number of failure free randomly selected test executions needed to certify required reliability. If all n_1 executions succeed, testing can stop and software reliability can be certified at the required level. Otherwise, a failure is observed at execution s_1 ($s_1 < n_1$). In the light of evidence of one failure in s_1 executions, a number of further failure free executions, n_2, where $n_2 > n_1$, is determined. Test executions proceed and either n_2 executions succeed (and reliability is certified), or a failure is observed on demand $s_1 + s_2$. In the later case, the testing continues. Note that if the program does not have the required reliability, testing may continue forever. In a sense, this technique is similar to Musa's reliability certification charts, but the reasoning that leads to accept or continue testing decisions is different. Readers interested in understanding the details of this approach are encouraged to read [14, 32, 33].

Despite all this work, our reading of the literature is that we are the first to discuss early stopping rules based on randomized search of formal models.

2.5 Other Non-Exhaustive Search Strategies

Our work might be characterized as the application of heuristic search to formal methods. The intuition behind heuristic search is that expert knowledge can solve problems faster than brute force methods. For example, one method is to maintain a "frontier" list of "promising" op-

tions. When new options are found, they are assessed via a heuristic "distance function" that guesses how close the new options are to the target solution. Options are added to the frontier sorted by the score assigned to them by the heuristic, and all but the top N options are ignored.

Experiments with this approach have yielded mixed results. Model checking has been optimized via such a heuristic search [18] but, in personal communication with us, the authors of that work report that the implementation was surprisingly complicated.

Others have used an *assume-guarantee* approach to reduce the running cost of model checking [47]. With this approach there are two properties specifications, those assumed, which are written into the software model, and those to be verified. By carefully choosing which properties go into each category, the analyst can control the size of model—and therefore the running time.

Our claim is, for procedures that exhibit the saturation effect, complex search strategies are not required, since random search would quickly find all that there is to find. Further, the simplicity of our implementation recommends it over (e.g.) approaches cited above.

2.6 Testability

The idea of saturation effects in software testing is related to the idea of *testability*. In a sense, we are using random search on formal models as a way of determing which are highly testable (i.e., those that exhibit a saturation effect) and which are difficult to test (i.e., those that do not exhibit saturation).

Testability has been defined more formally elsewhere: 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 [52], and later Bertolino and Stringini [5] clarify this definition, 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 the basis to draw inferences on system verification based on test results. According to Hamlet and Voas [21], 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 [5] 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 rather informal notion of testability, based on whether or not we see a saturation effect, is not inconsistent with the definitions above. If (a) a random search procedure over a NAYO graph reveals many unique reachable nodes in the model quickly and if (b) some of these nodes contain faulty logic, then those faults must be exposed. Note that when the search reaches saturation, 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.

3 Formal Models

Our experiments in random search over formal models are based on *communicating finite-state machines* (or FSMs) translated to a type of AND-OR graph (the compact NAYO graph representation mentioned in section 1). Finite-state machines are commonly used to formally model concurrent software systems. By translating finite-state models into NAYO graphs, we are able to represent the same information in a much smaller space. This section describes the automatic translation process.

3.1 FSMs

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 (including symbols representing states in other machines and symbols representing messages passed between machines).
- $\delta : Q \times B \longrightarrow Q \times B$, where B is a set of zero or more symbols from Σ, is the transition function.

Figure 3 shows a very simple communicating FSM model. States are indicated by labeled ovals, and edges represent transitions. Edges are labelled: *input* / *output*. Here are some of the parameters listed above, with their values for the model shown in Figure 3:

- Set of FSMs $S = \{M_{\mathsf{A}}, M_{\mathsf{B}}\}$.
- FSM $M_{\mathsf{A}} = (Q_{\mathsf{A}}, \Sigma_{\mathsf{A}}, \delta_{\mathsf{A}})$.
- Set of states $Q_{\mathsf{A}} = \{\mathsf{A1}, \mathsf{A2}\}$.
- Set of input / output symbols $\Sigma_{\mathsf{A}} = \{\mathsf{m}$ (message), $\mathsf{B2}$ (state from another machine)$\}$, etc.

Such FSMs can be used to formally model both a system and the properties of that system. By compiling *both* the

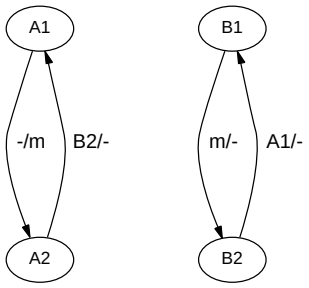


Figure 3. A system of two communicating FSMs (the machine on the right is referred to below as "M_A," the machine on the left as "M_B"; "m" is a message passed between the machines).

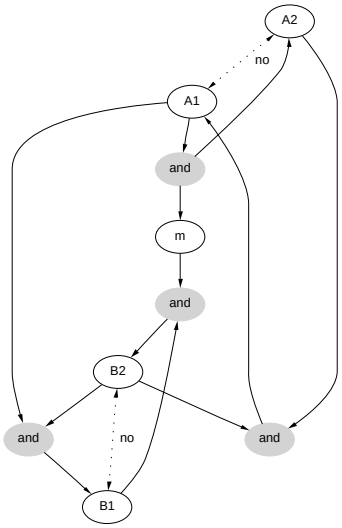


Figure 4. NAYO graph equivalent to FSM shown in Figure 3 (NO-edges dotted, AND-nodes shaded).

properties and system into FSMs, formal verification and heuristic testing reduces to the issue of *acceptance*. We say that the larger FSM representing the program *accepts* the smaller FSM representing the properties if the latter can be found within the former. If so, then the properties can be reached within the program.

It is usual practice to negate the desired properties before testing for acceptance. If acceptance of the negated properties, i.e., failure can be shown, then the path found by the model checker leading to the failure can be returned as a *counter-example* to the property, showing exactly how the program can fail.

3.2 NAYOs

Figure 4 shows an AND-OR graph equivalent to the communicating FSM model shown in Figure 3. We call this type of AND-OR graph a NAYO [38] since it contains the following features:

- A set N of undirected NO-edges connecting incompatible (logically *inconsistent*) 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.

Figure 5 shows the procedure used to automatically translate from a communicating FSM model (e.g., Figure 3), to a NAYO graph (e.g., Figure 4). 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 (Figure 5, line 5) + nk OR-nodes (line 3) = $O((m+n)k)$ nodes.

```
1: for (each finite-state machine) do
2:    for (each state) do
3:       Make an OR-node; connect it with a NO-edge to each
         OR-node representing another of this machine's states.
4:    for (each transition in this finite-state machine) do
5:       Make an AND-node;
6:       Make current state a YES-edge parent of the AND-node;
7:       Make input(s) (a) YES-edge parent(s) of the AND-node;
8:       Make next state a YES-edge child of the AND-node;
9:       Make output(s) (a) YES-edge child(ren) of the AND-node.
```

Figure 5. Automatic translation procedure from FSMs to NAYOs.

- $4mk$ YES-edges (lines 6-9) + $(n/2)(n-1)k$ NO-edges (lines 3-4) = $O((m+n^2)k)$ edges.

An 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 [27]. To give a more concrete idea of the space saved by using a NAYO graph, Table 1 compares the FSM composite to the NAYO graph for several models. Three of these models will be discussed further below.

Unfortunately the problem of finding all consistent assignments in a NAYO graph is NP-complete, as mentioned above and shown in [45]. So there is a tradeoff here: with a model checker, we can quickly search an exponentially large composite FSM; with our NAYO graph technique we can represent the information in a small space but require exponential time to exhaustively search it. The focus of

Model	States in FSM Composite (upper bound)	Nodes in NAYO Graph
Dekker 2-Process Mutual Exclusion Model from [19]	2,304	40
TCP Protocol Model from [48]	2,467	84
SCR Specification Model from [4]	1.05×10^{7}	73
Large Randomly Generated Model	2.65×10^{178}	4,007

Table 1. Size comparison of NAYO and FSM composite for a range of models.

model checking research has been to reduce the number of states required by the finite-state composite. Our NAYO graph scheme addresses the state *space* explosion problem, but creates a new problem, since exhaustive search requires exponential *time*. We address this NAYO time problem by using a partial (not exhaustive) random search.

4 Random Search

Our random search test scheme runs over an abstraction of the program, and so we do not have to explore the actual execution pathways of the programs normal operation. There is therefore a change of false negative—our method will find errors after exploring more behaviors of the program than seen in actual operation. However, given the saturation effect, there is *less chance* of a false positive, since our method over-samples the space of possible behaviors.

We say *less chance* rather than *no chance* for the following reason: if the program encounters input conditions that fall out of the range used in our random tests, then our search may not find the errors that result from those real-world inputs. In this regard, our technique is no different from any other method that draws conclusions based on selected input conditions. Nevertheless, our random search approach offers some advantages over other test methods. In this *two model method*, a very large space of inputs can be explored. And given that such formal models exhibit a saturation effect, our approach is not likely to be prohibitively expensive.

The rest of this section describes our random search method for NAYO graphs. This search is designed to solve the following problem:

- Given some (not necessarily consistent) input set of OR-nodes ...
- Find an output set implied by (and consistent with at least part of) the input, and make this output set as large as possible.
- Using this output set as input for the next iteration, find the next output set.
- Ultimately, we want a series of internally consistent output sets, each of which implies the next and is as large as possible.

Ideally each output set contains an OR-node for a state in each of the finite-state machines from the original model—if so, that output set is equivalent to one of the states in the composite FSM that would be searched by a model checker (and we have found it without explicitly constructing the composite FSM). But in general, because the random search is not exhaustive, it may not tell us quite as much as a more time- (and / or space-) consuming technique; that is, each output set will constitute only a *partial* description of a state in the composite.

The properties we check for are also partially defined states in the FSM composite that would be searched by a model checker. More complex temporal properties involve relationships between states, so to check for these using our approach we would search for relationships between *partially* defined states.

Figure 6 shows our NAYO random search procedure. The core ideas of the algorithm are *time-copies*, *random selection*, and *waitValues*. These terms are defined as follows.

Time Copies: The algorithm's view of time is that the NAYO graph is copied up to MAX times and nodes at time i connect to nodes at time $i+1$ via a *temporal linking policy*. Our policy is to use the outputs reached in one time copy as the inputs to the next time copy. Note that *time copies* is a conceptual view only. Figure 6 actually executes over a single time copy (with the appropriate indices being reset when the reasoning mores time i to time $i+1$).

Random Selection: The algorithm runs through a queue of found nodes and checks the children of each. If a child is consistent with the current output set, it is added to the queue—*inconsistency* would be indicated in the NAYO graph by a NO-edge between the child and some node in the current output set. This process continues until the queue is empty. Nodes are added to the queue in a random order (see lines 11, 28 and 32) The order in which nodes end up in the queue determines which will be popped first, and that determines which node's children via NO-edges will be disqualified first. In effect, the search flips a coin when it must decide between two incompatible paths.

WaitValue: Each time a node is reached its waitValue is decremented. Our NAYO traversal does not carry on the children of a node unless the node is waiting on nothing, i.e., its waitValue is zero (see line 27 of the algorithm). The waitValues are initialized as follows:

```
1: struct node {
2:   yesList (list of children via YES-edges)
3:   noList (list of children via NO-edges)
4:   type {AND,OR}
5:   disqualified (integer ≥ 0)
6:   waitValue (integer ≥ 0)
7:   found (integer ≥ 0)  }
   Initialize the graph:
8: for (∀ nodes n) do
9:   if (n.type = OR) then
10:    if (n ∈ input) then
11:      n.waitValue ← 0; Q ← n at random index.
12:    else if (n ∉ input) then
13:      n.waitValue ← 1.
14:  else if (n.type = AND) then
15:    n.waitValue ← |parents of n|.
   The main search procedure:
16: time ← 0.
17: while (time ≤ MAX) do
18:   while (Q ≠ ∅) do
19:     n ← pop(Q).
20:     if (n.disqualified ≠ time) then
21:       n.found ← time; n.disqualified ← time.
22:       for (∀ n' ∈ n.noList) do
23:         n'.disqualified ← time.
24:       for (∀ n' ∈ n.yesList) do
25:         if (n'.waitValue > 0) then
26:           n'.waitValue ← n'.waitValue − 1.
27:           if (n'.waitValue = 0) then
28:             n'.found ← time; Q ← n' at random index.
29:   for (∀ n) do
30:     if (n.type = OR) then
31:       if (n.found = time) then
32:         Q ← n at random index.
33:       else if (n.found ≠ time) then
34:         n.waitValue ← 1.
35:     else if (n.type = AND) then
36:       n.waitValue = |parents of n|.
37:   time ← time + 1.
```

Figure 6. Random search procedure for NAYO graphs.

- The waitValue of known inputs is zero.
- The waitValue of OR-nodes not included in the input set is initialized to 1, i.e., reaching any of the parents of an OR-node enables traversal to that node's children.
- AND-nodes' waitValues are initialized to their number of parents, i.e., all the parents of an AND-node must be reached before traversing on to the AND-node's children.

These waitValues are initialized in lines 8-15.

In the rest of this section we discuss the details of Figure 6. First of all, note that only OR-nodes may be part of the input set, because our translation procedure (Figure 5) creates only OR-nodes for states and messages in the original finite-state machines. Next, the *disqualified* field marks nodes known to be inconsistent with the set believed true at the current time (line 23). These are nodes reached via NO-edges; nodes are also disqualified after they are processed to prevent redundant searching at the current time (line 23). The *found* field marks nodes believed true at the current time.

The important part of the search occurs between lines 18 and 28. We begin with the input set of nodes in the queue. A node is removed (line 19); if it has already been disqualified, it is ignored (line 20). Otherwise, we mark it *found* and disqualify it so it will not be processed again (line 21). We then proceed to disqualify all of its children via NO-edges (line 23) and then explore its children via YES-edges. For these, we decrement waitValues, and if the waitValue of a child node becomes 0 (we now believe the node is true at the current time), we mark it *found* and put it in the queue at a random index. Lines 29–36 prepare for the time $i + 1$. Nodes found at time i are put into the queue for input to time $i + 1$ (line 32), and all other nodes are reset to their initial waitValues (lines 33–36).

From lines 17-18 and lines 22 and 24 of the algorithm, we see that it is linear on the maximum size of the queue, multiplied by some linear factors (the maximum time ticks used in the search, the maximum number of no-edges per node, and the maximum number of yes-edges per node). Since a node can only be placed in the queue once per time tick, the algorithm is $O(n)$, where n is the number of nodes in the NAYO.

This random search has the advantage of linear-time execution, but the theoretical disadvantage of incompleteness, since its random search may miss some faults. However, as the case studies in the next section show, for a given range of real-world and artificial models the incompleteness problem did not occur in experiments.

5 Random Search Case Studies

This section offers several case studies with our random search. For the case studies discussed here, random search was observed to be *adequate* (able to find errors before the saturation plateau) and to exhibit *flat plateaus* (saturation—therefore early stopping rules are supported).

Figure 7 shows Dekker's solution to the two-process mutual exclusion problem written in Promela (from [19]), which is the input language used with the model checker SPIN [26]. Promela has been designed to look like a high-level programming language, but represents communicating FSMs. SPIN is capable of automatically generating the finite-state machine version of a Promela model; Figure 8 shows the model from Figure 7 in this form.

Figure 9 shows the result of a series of random searches on a NAYO graph representing the finite-state model of Dekker's mutual exclusion solution from Figure 8. Each plot shows ten trials covering a range of MAX time values; for each trial the search in Figure 6 was repeated many times, each time with a random set of inputs, keeping track of the total OR-nodes processed (y-axis) and the unique

```
#define true  1
#define false 0
#define Aturn false
#define Bturn true

bool x, y, t;

proctype A() { x = true; t = Bturn;
               (y == false || t == Aturn);
               /* critical section */
               x = false }

proctype B() { y = true; t = Aturn;
               (x == false || t == Bturn);
               /* critical section */
               y = false }

init { run A(); run B() }
```

Figure 7. Dekker's solution to the two-process mutual exclusion problem, as Promela from [19].

```
proctype A
  state 1 -> state 2 => x = 1
  state 2 -> state 3 => t = 1
  state 3 -> state 4 => ((y == 0) || (t == 0))
  state 4 -> state 5 => x = 0
  state 5 -> state 0 => -end-

proctype B
  state 1 -> state 2 => y = 1
  state 2 -> state 3 => t = 0
  state 3 -> state 4 => ((x == 0) || (t == 1))
  state 4 -> state 5 => y = 0
  state 5 -> state 0 => -end-

proctype init
  state 1 -> state 2 => (run A())
  state 2 -> state 3 => (run B())
  state 3 -> state 0 => -end-
```

Transition inputs and outputs are in the column on the right. Inputs are enclosed in paranthesis, e.g., `((y == 0) || (t == 0))`; outputs are not, e.g., `x = 1`. For processes A and B, state 4 represents the area marked *critical section* in the Promela model.

Figure 8. Figure 7 model as finite-state machines output by SPIN.

OR-nodes reached (x-axis) during that trial. In order to show that our random search is capable of finding a fault, we have added to our NAYO graph a Boolean variable called *safe*, which is initially true and becomes false if proctype A and proctype B are ever simultaneously in state 4 (the critical section). We have also added the equivalent of the following transition to proctype A(), allowing it to go directly into its critical section without checking variables y and t:

```
state 3 -> state 4 => (true)
```

The searches shown in Figure 9 are typical of hundreds of experiments. In every fault-free Dekker model search, we quickly find all but one of the OR-nodes in the NAYO graph (23 of 24)—we never find the node *safe* = *false*. In the model with the fault, we quickly find every node (all 24), including the node representing the fault.

Figure 10 shows search results for the more complex TCP protocol finite-state model from [48]. This model consists of two larger FSM's (larger than FSM's in the Dekker model above), a client and a server. We see that here the search requires more time and is capable of finding a smaller percentage of the OR-nodes in the NAYO graph (there are about 40 OR-nodes in the NAYO graph; here we find 27). So the saturation effect is more pronounced in the Dekker model than the TCP model. We infer from this that the Dekker model is in some sense more testable than the TCP model.

Figure 11 shows search results for randomly generated models typical of thousands of experiments. Here we compare a model with size and structure like the Dekker model (a few small individual finite-state machines and no consumed message inputs) to a model like TCP (two large state machines and transitions triggered by approximately 20 different possible messages). Both plots show the results for twenty trials covering a range of MAX time values. As expected, the saturation effect is more pronounced in the Dekker-like model.

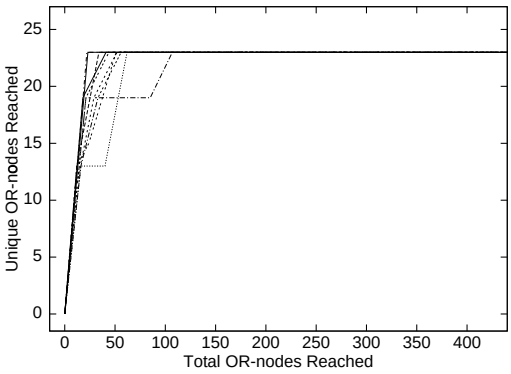
In Figure 12 we show search results for a very large model generated at random. This model has 250 individual finite-state machines with an average of about 6 states each and 1,455 local states in total. The size of its equivalent finite-state machine composite would be bounded at 2.65×10^{178} states. This is well beyond the capability of model checking technology (10^{120} states according to [11]).

6 Discussion

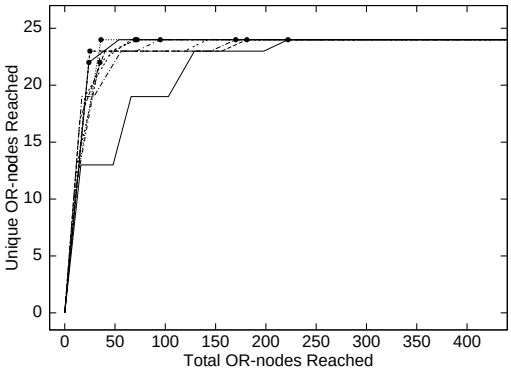
We have explored the merits of replacing an intractable complete search of formal models with an incomplete random search over formal models converted to NAYO graphs. Such incomplete searches are faster than complete search and require less memory.

Theoretically, such random searches suffer from the problem of false positives, i.e., reporting that no faults are present when further assessment would have found them. This theoretical problem did not show up in practice. Our random search was observed to be adequate (we found faults before the saturation plateau) and to exhibit flat plateaus (further search after the saturation point is not required).

A significant advantage of our scheme is the size of the formal models that can be tested. Figure 12 shows that our method can handle models of a size that would defeat most

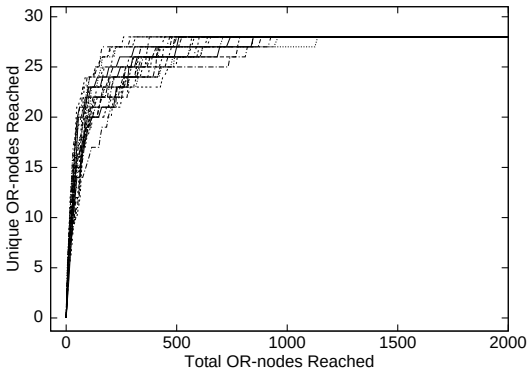


The correct version of the Dekker model.



Dekker model with an error added; dots show where in a particular trial the error node was reached.

Figure 9. Dekker's two-process mutual exclusion solution: random search results



Twenty trials covering a range of MAX time values; as before, in each trial the search is repeated many times, each time with a different random input set.

Figure 10. TCP model from [48]: random search results.

	Complete Search of NAYO Graph	Complete Model Checking Search	Partial Random NAYO Search
Memory	Small—$\mathrm{O}(n^2)$	Large (exponential)	Small—$\mathrm{O}(n^2)$
Time	Slow (exponential)	Linear on Memory	Linear on Input
Remarks	Complete	Complete	Surprisingly little incompleteness

Table 2. Comparison of Model Checking and Partial Random NAYO Search.

model checkers.

Another major advantage of our approach over traditional model checkers is the simplicity of our scheme. Figure 5 and Figure 6 show most of the details of our approach.

Table 2 compares the resources and capabilities of model checking, complete (exhaustive) search of NAYO graphs, and partial testing of NAYO graphs using random search. This table clearly captures the motivation for work on partial random NAYO search: we solve the exponential memory size requirement problem by using a NAYO representation, and we solve the exponential time problem of exhaustive search by using a partial random search. Further, the random NAYO search was not only found adequate (since it could find errors), it was also determined to be surprisingly effective (since the observed plateaus were very flat).

We propose the following early stopping rule for testing of formal models:

- After representating FSMs as NAYOs, track the number of unique nodes found during random search.
- Stop when that number plateaus.
- If that plateau never occurs, then abandon random search and use more complete methods such as model checking or theorem proving.

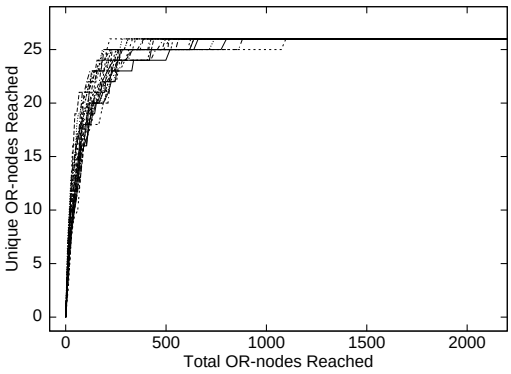
For domains that exhibit the saturation effect, model checking and/or theorem proving may not be required. Based on our experiments, two average case mathematical analyses [38, 41], and a literature review [35], we believe the saturation effect is common. We further speculate that random search will be sufficient in many domains. Hence, early stopping often will be possible. Providing more evidence in support of this claim the the topic of our ongoing research efforts.

References

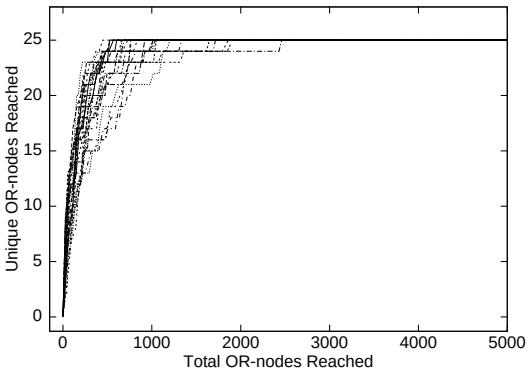
[1] IEEE glossary of software engineering terminology, ANSI/IEEE standard 610.12, 1990.

[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.



Random search of a Dekker-style model (several small FSMs).



Random search of TCP-like models (two large FSMs).

Figure 11. Search results for two random models.

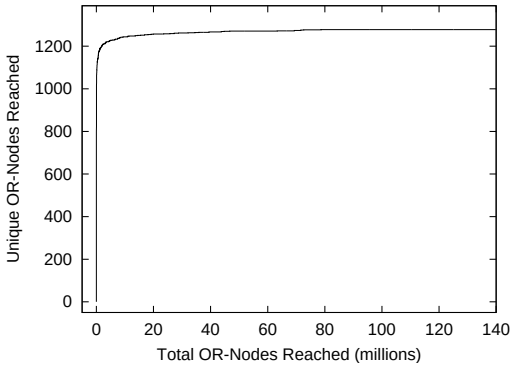


Figure 12. Random search of very large models.

[4] Bemina Atanacio. Modeling the Space Shuttle Liquid Hydrogen Subsystem. Technical report, The Software Engineering Institute, Carnegie Mellon University, 2000.

[5] 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.

[6] 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.

[7] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, 1980.

[8] E. Clark and T. Filkorn. Exploiting symmetry in temporal logic model checking. In *Fifth International Conference on Computer Aided Verification*. Springer-Verlag, 1993.

[9] E. Clark and D. E. Long. Compositional model checking. In *Fourth Annual Symposium on Logic in Computer Science*, 1989.

[10] E. Clarke, O. Grumberg, and D. Long. Verification Tools for Finite State Concurrent Systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency-Reflections and Perspectives*, pages 124–175. Springer-Verlag, 1993.

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

[12] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasarenu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings ICSE2000, Limerick, Ireland*, pages 439–448, 2000.

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

[14] B. Cukic and D. Chakrawarthy. Bayesian framework for reliability assurance of a deployed safety critical system. In *Proceedings of the 5th International Symposium on High Assurance Systems Engineering, Albuquerque, NM, November*, 2000.

[15] J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, 28:163–196, 1986.

[16] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE98: Proceedings of the 21st International Conference on Software Engineering*, May 1998.

[17] M. B. Dwyer, G. S. Avrunin, and J. Corbett. A system specification of patterns. `http://www.cis.ksu.edu/santos/spec-patterns/`, 1997.

[18] S. Edelkamp, A. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI Symposium on Model-based Validation of Intelligence*, 2001. Available at `citeseer.nj.nec.com/edelkamp01protocol.html`.

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

[20] P. Godefroid. On the costs and benefits of using partial-order methods for the verificiation of concurrent systems (invited papers). In *The 1996 DIMACS workshop on Partial Order Methods in Verificaition, July 24-26, 1996*, pages 289–303, 1997.

[21] 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.

[22] M. Harrold, J. Jones, and G. Rothermel. Empirical studies of control dependence graph size for C programs. *Empirical Software Engineering*, 3:203–211, 1998.

[23] C. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, January 2002. Available from `http://chacs.nrl.navy.mil/publications/CHACS/20022002heitmeyer-encse.pd%f`.

[24] C. Heitmeyer, B. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *International Symposium on Requirements Engineering, York, England , March 26-27*, 1995.

[25] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996. Available from `http://citeseer.nj.nec.com/heitmeyer96automated.html`.

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

[27] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990. Available at `http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html`.

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

[29] 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.

[30] Y. Ishida. Using global properties for qualitative reasoning: A qualitative system theory. In *Proceedings of IJCAI '89*, pages 1174–1179., 1989.

[31] 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`.

[32] B. Littlewood and D. Wright. Stopping rules for the operational testing of safety critical software. In *Proceedings of the 25th Conference on Fault Tolerant Computing (FTCS 25), Pasadena, CA, July*, 1995.

[33] B. Littlewood and D. Wright. Some conservative stopping rules for the operational testing of safety-critical software. *IEEE Transactions on Software Engineering*, 23(11):673–683, November 1997.

[34] A. Mackworth and E. Frueder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[35] T. Menzies and B. Cukic. Adequacy of limited testing for knowledge based systems. *International Journal on Artificial Intelligence Tools (IJAIT)*, June 2000. Available from `http://tim.menzies.com/pdf/00ijait.pdf`.

[36] T. Menzies and B. Cukic. When to test less. *IEEE Software*, 17(5):107–112, 2000. Available from `http://tim.menzies.com/pdf/00iesoft.pdf`.

[37] 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`.

[38] T. Menzies, B. Cukic, H. Singh, and J. Powell. Testing nondeterminate systems. In *ISSRE 2000*, 2000. Available from `http://tim.menzies.com/pdf/00issre.pdf`.

[39] 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://tim.menzies.com/pdf/01agents.pdf`.

[40] T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via 'topoi diagrams'. In *ICSE 2001*, 2001. Available from `http://tim.menzies.com/pdf/00fastre.pdf`.

[41] T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In *2nd International Workshop on Soft Computing applied to Software Engineering (Netherlands), February*, 2001. Available from `http://tim.menzies.com/pdf/00maybe.pdf`.

[42] 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.

[43] J. Musa. *Software Reliability Engineered Testing*. McGraw-Hill, 1998.

[44] J. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw Hill, 1987.

[45] D. Owen. Random search of and-or graphs representing finite-state models, 2002.

[46] A. Parkes. Lifted search engines for satisfiability, 1999.

[47] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *SPIN*, pages 168–183, 1999.

[48] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[49] 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.

[50] J. Singer, I. P. Gent, and A. Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.

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

[52] 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`.

[53] 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.