

Software Assurance by Bounded Exhaustive Testing

Kevin Sullivan¹, Jinlin Yang¹, David Coppit², Sarfraz Khurshid³, Daniel Jackson³

{sullivan, jy6q}@virginia.edu; coppit@cs.wm.edu; dnj@lcs.mit.edu; khurshid@csail.mit.edu

1. University of Virginia; 2. The College of William & Mary; 3. MIT

ABSTRACT

The contribution of this paper is an experiment that shows the potential value of a combination of selective reverse engineering to formal specifications and bounded exhaustive testing to improve the assurance levels of complex software. A key problem is to scale up test input generation so that meaningful results can be obtained. We present an approach, using Alloy and TestEra for test input generation, which we evaluate by experimental application to the Galileo dynamic fault tree analysis tool.

Categories and Subject Descriptors

D.2.4. [Program Verification]. D.2.5 [Testing and Debugging].

General Terms

Experimentation, Reliability, Verification.

Keywords

Bounded exhaustive testing, specification-based testing, automated test case generation, TestEra, formal methods, reverse engineering

1. INTRODUCTION

Assuring the trustworthiness of even modestly complex software programs remains a difficult, open problem of great importance. An approach that has been proposed is *bounded exhaustive testing* (BET) [32]: exhaustively testing all inputs up to a given complexity or size. The underlying hypothesis is that, in practice, any given failure mode is likely to manifest itself on some small input, and testing all small inputs thus suffices to reveal these failure modes.

The primary contribution of this paper is an empirical test of the feasibility and potential utility of bounded exhaustive testing (BET) on a reasonably complex, production software system. We adapted and applied BET to test the existing Galileo tool [8,15,42] for dynamic fault tree (DFT) modeling [13,14,16], which is used in the reliability analysis of complex engineering systems.

We had to perform three major technical tasks to enable the use of BET. The first was to write a formal specification of the abstract syntax and semantics of DFT's. The second was to develop a test oracle implementing the semantic mapping from DFT's to reliabilities. The third was to mitigate scalability limitations of the tool used to generate test inputs from the DFT syntax specification.

Our previous work [9,10,11] addressed the first two issues: the cost-and technical-effectiveness of *selectively* reverse engineering

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA'04, July 11-14, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-820-2/04/0007...\$5.00.

Galileo to a formal specification of its core analysis capabilities, and producing an implementation, designed for verifiability not efficiency, which we used here as a test oracle. This paper takes the step, envisioned in the earlier work [10], of using the specification to drive BET. We report on our use of TestEra [32], whose inputs are Alloy specifications [23], to generate all non-isomorphic fault tree input structures up to the computational limits of TestEra and the oracle. We measured utility in terms of unknown failure modes revealed, and cost in terms of labor required.

The challenges encountered in this part of the work were, first, to generate and run test inputs within sufficiently large bounds to get meaningful results, and, second, to diagnose large numbers of observed failures. The first challenge arose from limitations of the test input generator and test oracle. The second challenge arose because, in exhaustive testing, many inputs reveal the same failure.

This paper focuses on the first problem. We started by translating our earlier Z [40] specification of DFT syntax into Alloy for use as an input to TestEra. When we first ran TestEra it was unable to generate inputs beyond very small bounds due to excessive memory requirements. We developed techniques of specification abstraction and tightening to extend its range. In the end we generated millions of non-isomorphic test input structures and were able to run hundreds of thousands of test cases, revealing several previously unknown failure modes.

Traditional testing subjects a system to orders of magnitude fewer tests, whose construction is labor intensive and unreliable. To our knowledge, this work is the first to evaluate BET for a system with input structures and computational procedures as complex as those for DFT's. Although we address only Galileo and have no evidence that our results generalize beyond Galileo-like systems, the market for a pragmatic, automated, formal approach is potentially great, e.g., improving software assurance of legacy systems. Testing cyber-infrastructure for vulnerabilities is a possible application.

The rest of this paper is organized as follows. Section 2 provides background on specification-based testing. Section 3 describes our approach. Section 4 describes impediments to test input generation encountered and how we overcame them. Section 5 explains how we used the approach to test Galileo. Section 6 reports our experimental results and findings. Section 7 reflects on our findings. Section 8 discusses related work. Section 9 concludes.

2. SPECIFICATION-BASED TESTING

We begin with a review of specification-based testing and test selection criteria. Specification-based testing [2] uses the target program's specification, usually written in a formal specification language, to generate a set of program inputs and to judge the resulting outputs. Tools such as Korat [3] and TestEra [32] have been developed to automate this process. It is claimed that specification-based testing can improve the effectiveness of testing by reducing the cost and increasing the chance of detecting bugs.

Real systems often have infinite input spaces, making exhaustive testing impossible. As a result, many well-known test selection criteria [2] have been developed to select test cases from an input space to achieve limited confidence. The statement coverage criterion, for example, requires that test cases be chosen to execute each statement of a program at least once. Harrold [19] points out that more research is needed to provide evidence of the effectiveness of the test selection criteria in revealing faults.

More recently, bounded exhaustive testing criterion has attracted attention as an adequacy criterion. It requires *exhaustive* coverage of the input space up to certain bounds on input size. Previous work has demonstrated the feasibility and effectiveness of bounded exhaustive testing on some small-scale systems using the TestEra tool [32]. Prior to this experiment, the largest system on which the technology had been evaluated was the Intentional Naming System [32,38], with 2000 lines of Java code. Evidence of applicability to larger-scale systems with structurally more complex input spaces was absent. This paper helps to fill that void with a study of the potential utility of TestEra for bounded exhaustive testing of a complex tool.

TestEra itself is a framework for automated specification-based testing [2] (with an orientation to Java programs, in particular). To test a method, the user provides a specification that consists of a precondition (which describes allowed inputs to the method) and a post-condition (which describes the expected outputs). TestEra uses the precondition to automatically generate a test suite for all test inputs up to a given bound; a test input is within a bound of k if at most k objects of any given class appear in it. TestEra executes the method on each input, and uses the post-condition as a test oracle to check the correctness of each output.

TestEra specifications are first-order logic formulas. As an enabling technology, TestEra uses the Alloy toolset. Alloy [23] is a first-order declarative language based on sets and relations. The Alloy Analyzer [24] is an automatic tool that finds instances of Alloy specifications, i.e., assignments of values to the sets and relations of the specification that make the specifications constraints evaluate to true. The analyzer finds an instance by: 1) translating the Alloy specification into a Boolean satisfiability formula, 2) using an off-the-shelf SAT solver to find a solution to the formula, and 3) translating the solution back into sets and relations. The analyzer can enumerate all instances (within a given bound) using a SAT solver that supports enumeration, such as mChaff [34] or relsat [1]. The analyzer generates complete assignments: if the underlying SAT solver generates a solution with "don't care" bits, the analyzer grounds these bits out, i.e., systematically replaces them with 0's and 1's.

TestEra translates Alloy instances into test inputs that consist of Java objects. Some inputs are isomorphic, i.e., they only differ in the identity of their objects. Two linked lists, for example, containing the same elements (up to isomorphism) in the same order but whose buckets are different nodes will be isomorphic, since no code will be able to tell them apart. Considering only non-isomorphic inputs reduces the time to test the program, without reducing the possibility of detecting bugs, because isomorphic test inputs form a *revealing sub-domain* [45], i.e., produce identical results¹. The analyzer has automatic symmetry breaking [39] to eliminate many isomorphic inputs. TestEra users

can also exploit support for total orders to guarantee generation of exactly non-isomorphic inputs [25].

Initial case studies with TestEra focused on checking Java programs. TestEra exposed bugs in a naming architecture for dynamic networks [26] and a part of an earlier version of the Alloy Analyzer [32]; these bugs have now been corrected. TestEra was also used to systematically check methods on Java data structures, such as from the Java Collection Framework [43].

3. OUR APPROACH

First, we selectively produce a formal specification of the parts of a system viewed as most critical. The result is a mathematically precise specification of the required behavior of that part of the system under study.

Galileo was an existing system, so this activity amounted to reverse engineering. We studied the code and other documentation, worked with domain experts, etc. We ended up with a specification of what the program should have been doing, which was slightly different from what it was actually known to be doing.

Second, from the validated specification we derive two artifacts. The first is an oracle designed for verifiability, not efficiency. The result is in general an oracle that is unacceptably inefficient for inputs that arise in practice but that is still adequate for *small* inputs. In the future, we may explore executable specifications to avoid having to derive separate oracle programs. We emphasized verifiability of the oracle code by preserving and inspecting, as far as possible, an isomorphism between the specification and its C++ code (in modularity, identifier names, data structures, etc) [7].

The second artifact derived from the specification is the input to a specification-based test input generator. If the overall specification is structured appropriately and written in the same language as that taken by the generator, this step involves no more than extracting the relevant parts. In our case, the main specification was written in Z [40], so the extraction of the input-space characterization (DFT syntax) required a translation from Z to Alloy.

Third, we present the generated test cases as inputs to the legacy system and oracle, and capture and compare the outputs.

Fourth, when the outputs disagree, we trace the cause of the disagreement to faults in the legacy system, formal specification, generator input (if different from the specification), or oracle.

An interesting phenomenon arises here. A given fault is often revealed by a very large number of test cases. The problem then is to partition a potentially huge number of observed failures into a small number of equivalence classes by causative fault.

We explored an iterative approach to address this problem. To begin, we select a single test case that reveals a fault and ask domain experts to provide a correct output. The smallness of inputs is advantageous here: it makes manual computation of the correct answer easy. The hand-computed answer typically exposes the location of the fault: system, oracle, or specification.

If the system produces an incorrect answer, a bug has been found. If the oracle produces a wrong answer, the problem is either in the oracle implementation or the specification. If the problem is in the system or oracle, a decision to fix the bug or move on is made, based on economic concerns. If the decision is to postpone the fix, we modify the generator specification to *mark* with a comment, "*expect-failure*," all cases known to exercise the fault, thus

¹ We assume that object identities are used for comparisons only.

documenting the fault and causing our test harness to skip such cases for an effectively much smaller test. For all the cases where we decided to postpone a fix, it was easy to identify all fault trees that would fail as a result of the given fault. For example, we could easily mark generated fault trees that contained components with inappropriately high failure rates; such failure rates would cause the solver to compute invalid results. There is a potential risk in deferring repairs: a marked input could also reveal a second fault if the first fault is repaired and the test case is executed again.

Once all expected-failure inputs have been marked, we retest. Additional failures at this point can only be attributable to other faults. This process is iterated until no more faults are revealed. If the fault is found to be in the specification, we correct it and update the generator specification accordingly.

This whole process is iterated. Each iteration increases the bounds on test case generation. Having removed or masked all faults revealed by inputs up to size n , we try to generate inputs for size $n+1$ and start again. We pursue this course until computational resources (of the generator or oracle) are exhausted.

4. TECHNICAL CHALLENGES

Three elements limit the scalability and thus effectiveness of our approach: the test generator, the oracle, and the cost of handling large numbers of observed failures. First, the test case generator cannot generate inputs beyond certain size bounds. It was unclear what the bounds were or whether they were sufficiently large to permit a really useful test set to be produced. Second, the oracle is designed for verification, eschewing optimization. An oracle might be (and in our case was) incapable of running quickly enough beyond certain relatively small input sizes. We have already discussed the problem posed by large numbers of observed failures, and how we addressed it.

The main challenge is to loosen the bottleneck at the test generator. Direct use of specifications produced by reverse engineering proved inadequate: TestEra was unable to generate tests beyond very small bounds. To extend the bounds, we devised several strategies. The most interesting, which we now discuss, was *separation of generation concerns* through a combination of *specification abstraction* and *abstract test case post-processing*.

Separating generation concerns means offloading from the generator to a post-processor the handling of aspects for which the unique abilities of the generator are not needed. For example, aspects of fault trees include their tree-like structures, the kinds of *gates* and *basic events* at the nodes, and scalar values at basic events. There are a number of complex global constraints which must be satisfied for a fault tree to be valid, such as lack of cycles. TestEra is well-suited for generating complex *structures*. Having it generate from our original specification, however, required it to enumerate identical structures differing only in gate and basic event types and scalar values. TestEra so configured was not able to generate tests to satisfactory bounds. It would terminate abnormally, out of memory, even before traversing the space of fault trees of three events (e.g., one gate, two basic events).

Our approach to this problem is to abstract details from the specification that need not be handled by TestEra, replacing them with placeholders, in order to focus the generator on the hard problem of generating *abstract fault tree (AFT) structures*. Then we use a post-processor (in this case some simple Perl scripts) to

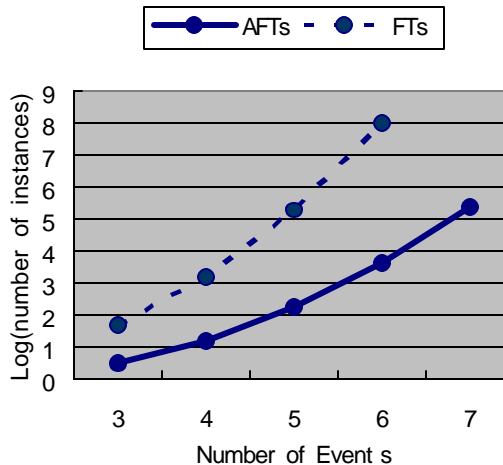


Figure 1. Comparison of Number of FT's and AFT's

generate *concrete fault trees (FT's)* from the abstract ones by systematically substituting all possible values at the placeholders.

In a first effort, we separated out the handling of scalar values. In a second phase, we abstracted out the gate types at the nodes, significantly extending the range again.

We can now generate all concrete trees with up to *six* events and all abstract trees with up to *seven*. (See Table 1 for details.) Within this scope significant interactions among the features of the fault tree language can arise, and it is just in these interactions where the most serious faults are expected to lie.

There are at least two benefits to this separation of concerns. First, the generator is able to exhaustively generate to a larger bound because the abstracted specification is less complex than the original, with the number of states to explore much smaller. Second, it is easier to validate the generation itself, since there are fewer abstract than concrete inputs, making it possible (for very small bounds) to inspect the inputs manually.

Figure 1 compares the number of fault trees and abstract fault trees generated as a function of the bound on the number of events (gates and basic events) in a tree. The X-axis represents the number of events; the Y-axis, the logarithm of the number of instances. The dashed line denotes fault trees; the solid line, abstract fault trees. The curve for AFTs is not only lower, allowing generation to a larger bound, but it also grows more slowly so that improvements in generation technology have more leverage. It was also much easier to verify the abstract generation results. For example, with a bound of four events, there are only sixteen AFTs. One can manually verify that all are generated correctly. There are, however, 1571 fault trees for the same number of events.

5. GALILEO: A CASE STUDY

To develop and evaluate our approach we used it to help verify a software system that we are developing for NASA. The system, called Galileo [8,10,15,42], allows reliability engineers to model and analyze the reliability of complex systems. It has been deployed for production use at NASA.

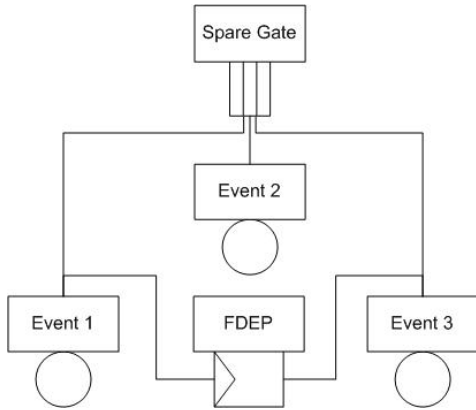


Figure 2. A dynamic fault tree

5.1. Dynamic Fault Tree Analysis

Dynamic Fault Trees (DFT) [4,14,44] is a reliability modeling notation that allows the engineer to model complex failures in systems having redundancy and other fault tolerance mechanisms. It supports modeling of uncovered failures, systems with multiple phases of operation, sparing behavior, and more.

A dynamic fault tree is a directed acyclic graph in which the internal nodes (gates) represent relationships among their inputs, and the leaf nodes (basic events) represent component failures or event occurrences. Gates can be either static or dynamic, depending on whether the order of failures of inputs to the gate is relevant. Dynamic fault trees can also model constraints between failures. For example, the functional dependency constraint ensures that a given failure in the model will cause the immediate, simultaneous failure of one or more dependent basic events. A sequence enforcing constraint ensures that certain failures do not occur out of order.

Figure 2 shows an example fault tree with a spare gate, a functional dependency, and three basic events. In this example, Events 1, 2, and 3 might represent components in the system. The spare gate is a dynamic gate that uses Event 1 until it is no longer in operation, then uses Event 2, then Event 3. When there are no spares available, the spare gate fails, as does the system. The functional dependency indicates that the failure of Event 1 causes the immediate failure of Event 3.

Fault trees can be used to compute a number of properties about the system being modeled. For example, given a fault tree, suitable statistical data on the failure of the basic events, and a system operation time one can compute the overall probability of failure.

Analyses are performed by converting fault trees into state-based or combinatorial representations [4]. Figure 3 shows a state machine for the example tree. The labels on the arcs represent stochastic transition rates. The left-most state is the initial state, in which all the basic events are operational and the spare gate is using Event 1. Transitions from states model basic event failures. For example, when Event 1 fails, the next state has both Event 1 and Event 3 failed (the latter due to the functional dependency),

and the spare gate using Event 2. The shaded state is a system failed state, as indicated by the failed status of the spare gate.

With certain qualifications (beyond the scope of this paper), such a state machine can be converted to a continuous-time Markov chain by replacing the labels on the arcs with the proper failure rates. One can then compute the probability of failure by solving the Markov chain and determining the probability of being in any state in which the top-level event in the tree is failed.

While the example shown is simple, several complications make it hard to develop an efficient, dependable software implementation. First, interaction between the different modeling constructs complicates the language semantics. Second, direct translation to Markov chains is swamped by combinatorial explosion even for relatively small trees. Galileo employs two major optimizations. First, it decomposes large trees into independent sub-trees when it can, solves the sub-trees independently, and composes the results to produce an exact result.

Second, Galileo supports two solvers: a *dynamic solver* that solves fault tree by translation to and analysis of Markov chain; and a *static solver*, by translation to a BDD [13]. The dynamic solver can solve most trees. The static is applicable only to trees without dynamic gates. For such cases, however, it is enormously more efficient. A crucial property, to which we will return in Section 6.2, is that the two solvers should produce exactly the same answers on inputs to which they are both applicable. What we did, then, in testing Galileo against the oracle was to run each input for *each* of the Galileo solvers whenever possible. We assumed that a discrepancy between the oracle and one of the solvers would clearly indicate a fault in the discrepant solver. As we discuss in Section 6.2, we were wrong in one especially interesting case.

5.2. Galileo

Galileo [8,15,42] is a tool for modeling and analyzing fault trees. It began as a research prototype for assessing the viability of using mass-market applications as components. Due in part to the success of this approach, NASA has supported more recent efforts to further develop the tool. In fact, NASA is currently using the tool to help model and diagnose faults in the International Space Station. As a result of these developments, the original weak dependability requirements of a research prototype are no longer appropriate.

Recent enhancements to Galileo have placed greater pressure on dependability. New features such as phased mission modeling and analysis, diagnostic decision trees, and sensitivity analysis complicate the software analysis routines considerably. As a result, the dependability of the DFT solver, even with respect to simple analyses, is of increasing concern.

For these reasons, we decided to focus our verification efforts on the core reliability analysis capability, avoiding for now reliability issues of the user interface and more sophisticated modeling and analysis capabilities. To this end, we created a simple, easily testable command-line version of the tool containing only the DFT solver, fault tree data structure, and a textual parser, totaling 10,680 of non-comment, non-blank lines of C++ code.

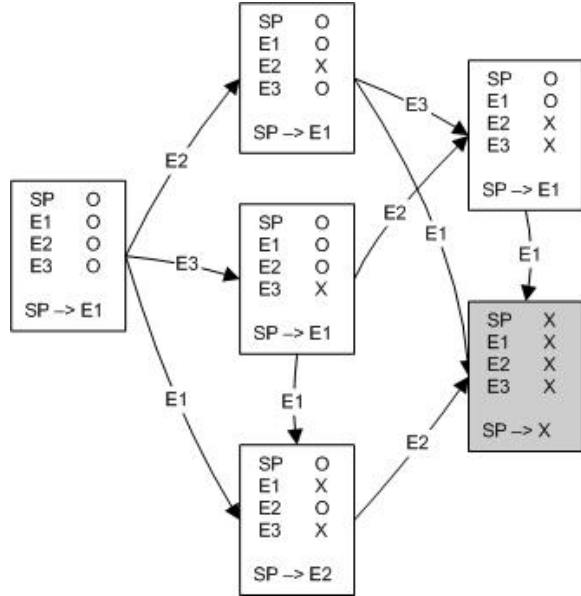


Figure 3. State machine for the example DFT

5.3. Nova Implementation as Test Oracle

Nova [9,10] is a prototype dynamic fault tree modeling and analysis tool. It was developed to evaluate the feasibility of combining formal methods and package-oriented programming to achieve both dependability and usability at low cost (albeit in this case at the cost of efficiency). As part of this effort, Coppit and Sullivan developed and validated a formal specification, written in Z [40], of the abstract syntax and semantics of dynamic fault trees. The specification was validated via technical reviews with domain experts and limited formal analysis [9,10,11] using Z/Eves [37].

Here, we leverage the previous specification and implementation effort. As with Galileo, we extract the DFT solver module and use it to implement a text-only command line solver program. Nova supports a variant of the Galileo DFT language which has been redesigned in key ways to improve its regularity and orthogonality characteristics. Unlike Galileo, Nova does not support phased missions, sensitivity analysis, or other more advanced capabilities.

The Nova solver was developed from the formal specification. While we did not perform a formal refinement to code, we did work to design an implementation that would be easy to verify by inspection [7]. In particular, we eschewed almost all optimizations in data representation, and we sought to preserve, to the extent possible, an isomorphism between specification and code. Each key abstraction in the specification, for example, is represented by a corresponding one in the code. The close correspondence eases verification by inspection, giving us more confidence in the use of the program as an oracle. On the other hand, the combinatorial explosion in mapping fault trees to Markov chains swamps Nova much more quickly than Galileo. For dynamic fault trees with fewer than 4 nodes, Galileo is about 5 times as fast as Nova. As trees grow in size, Nova becomes a bottleneck in testing.

5.4. TestEra as Test Case Generator

The Galileo fault tree solver takes as inputs strings in the fault tree grammar. For input generation, we wrote a concretization

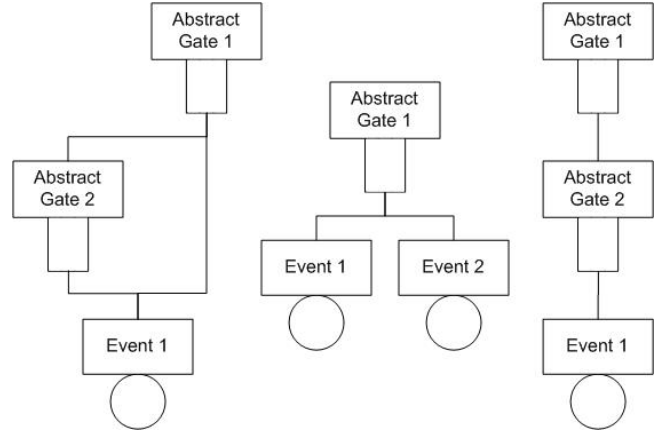


Figure 4. All AFTs with three events, no Seq's or FDep's

translation by hand using Java and Perl, to generate fault trees in this grammar from TestEra's output

As explained above, we used TestEra to generate only abstract inputs; concrete fault trees were obtained in an additional step with a Perl script replacing place-holders with all possible values from appropriate domains. For each abstract input, this step may generate a combinatorially larger number of concrete inputs. Prior uses of TestEra have involved data translations that also make use of a similar form of abstraction, but map each abstract input to one concrete input.

Even for small bounds, there are a very large number of fault trees. This necessitates generation to be restricted to exactly non-isomorphic trees to enable feasible enumeration and checking. Following Khurshid et al. [25], we manually added simple symmetry breaking predicates to enable such generation.

Even with symmetry-breaking, the number of inputs generated from our initial specification was infeasibly large. We tightened our specification to incorporate test purposes, such as generation of only those fault trees that represent a single connected component, and focused checking on the desired functionality (as explained earlier in this paper.).

6. RESULTS

For test generation we used a dual-CPU Pentium 3 at 1GHz with 1GB of RAM running Red Hat Linux 2.4.18-27.7.xsmp #1 SMP. The generator ran unsatisfactorily under Windows due to incompatible memory management requirements of the underlying SAT solver. The machine used to run test cases was a dual-CPU Pentium 4 at 3GHz with 1GB RAM and Windows XP Professional version 2002 SP 1.

As an example of the fault trees we generated, Figure 4 shows the three non-isomorphic abstract FTs generated with three events, and no dependency constraints (functional dependencies or sequence enforcements). A Perl script instantiates each *Abstract Gate* with any of the five types of gate: AND, OR, PAND, KofM, or SPARE.

We were able to generate all abstract fault trees with up to seven events and no dependency constraints; instantiate all concrete FTs with up to six events and no constraints; and generate all concrete FTs with up to *five* events, *at most one* FDep, and *at most one* Seq.

Table 1 summarizes the number of abstract and concrete trees. It took about 75 hours to generate all abstract trees with seven events,

Table 1. The number of DFTs and AFTs generated

Events	Seqs	FDEps	AFTs	DFTs
3	0	0	3	48
3	0	1	4	56
3	1	0	6	96
3	1	1	8	112
4	0	0	16	1,571
4	0	1	46	3,614
4	1	0	192	18,852
4	1	1	552	43,368
5	0	0	176	186,668
5	0	1	717	616,806
5	1	0	10,560	11,201,520
5	1	1	43,020	37,017,000
6	0	0	4,229	93,454,072
7	0	0	230,470	astronomical

of which there are over two hundred thousand. The number of concrete fault trees at this level is astronomical. At a scope of six events, there are over four thousand abstract trees and ninety three million concrete trees. We tested all concrete trees up to scope four, and to scope five with no constraints. That amounted to about two hundred fifty thousand inputs. We ran the production and oracle code, which took about a week, identified and diagnosed failures, and corrected or masked the faults. It would be impossible to run all trees with six events due to the inefficiency of the oracle. It took about 2 minutes to run a six-event input, so it would take 355 years to run all such FTs. In future work, we will investigate principled, verifiable optimization of the oracle as a strategy for extending the feasible testing bound. Testing revealed eight faults in Galileo, three in the specification, and three in the oracle. To make the results concrete, we describe an instance of each.

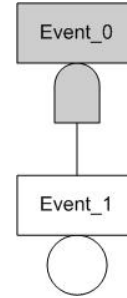
6.1. A Fault in Galileo

Figure 2 shows a test case that revealed a fault in Galileo. As discussed in Section 5.1, basic events Event 1, Event 2, and Event 3 must all fail for the spare gate to fail. If only Event 1 fails, the spare gate should remain operational for as long as Event 2 does. In the Markov chain for this tree, from the initial state in which no events have failed a transition for Event 1 failing should lead to a state in which Event 2 and the spare gate remain operational (the lower-most state in Figure 3). However, debugging showed the transition led to a state with Event 2 operational but not the spare gate (the system-level failure event). The resulting unreliability estimate for this model was thus higher than the correctly computed value. This fault had gone undetected for at least three reasons. First, the tool has still not yet seen extensive production use. Second, the fault is in the interaction of two more rarely used constructs (FDep and multiple spares). Third, there’s no easy way to see that the output is wrong.

6.2. A Fault in the Specification

One of the most interesting faults that our testing revealed was the omission of an important precondition from the specification. The fault both was revealed by N-version programming [29], and showed that we had also been “bitten” by one of its known risks.

Recall that we tested both the static and dynamic solvers against the oracle. We assumed that a discrepancy between the oracle and one of the solvers would reveal a fault in the other solver. That assumption turned out to be wrong.



```

toplevel Event_0;
Event_0 and Event_1;
Event_1 lambda=.01 cov=0 res=.5 repl= 2 dorm=.5;

```

Figure 5. The DFT that revealed a specification error

One equivalence class of tests revealed a discrepancy. Galileo’s dynamic solver, which translates fault trees to optimized Markov chains, agreed with the oracle, which translates them to un-optimized Markov chains; but Galileo’s static solver, which translates fault trees to BDD’s, disagreed with both. The test case was given to our domain experts with a request to verify that the static solver did not implement its specification.

To our befuddlement, our colleagues reported that the static solver calculated the correct answer according to published definitions. We then assumed that the dynamic solver and oracle must be exhibiting a common failure—a classic N-version programming problem. We were wrong here, too. We asked our experts to compute the Markov-based solution by hand. They reported that it, too, was computed correctly. The two methods were “known” to yield the same, exact, answers, but they didn’t.

A weekend’s inquiry by a leading domain expert finally produced an explanation. The journal article in which the equivalence of the two methods was demonstrated contained an easy-to-overlook condition: *provided that basic event failure probabilities are sufficiently small, the methods are valid and produce negligibly divergent answers.* (The article provided no definition of *sufficiently small*.) High failure rates would violate the statistical assumptions on which the methods are based. Figure 5 shows a very simple test case that leads us to detect this fault. *Event 1*, a basic event, has a large lambda value (e.g. failure rate), .01. This condition had been forgotten, in a sense. Our testing recovered it.

What happened? The Markov-based dynamic solvers (Galileo and oracle) agreed: an N-version failure on a common, undocumented precondition. The static solver was also subject to the specification fault but misbehaved in a different way: an N-version programming success based on algorithmic diversity. Galileo now uses a heuristic check and warns of high probabilities.

6.3. A Fault in the Nova Test Oracle

We also found a bug in Nova’s implementation of sequence enforcement constraints. Recall that a sequence enforcer ensures that events occur in a particular order. In Figure 6, the sequence enforcer states that *Event 2* will not fail before *Event 1* fails. The implementation should invalidate out-of-order failures. Nova erroneously omitted this check. Our approach to informal but principled derivation of oracles from specifications clearly remains subject to human error.

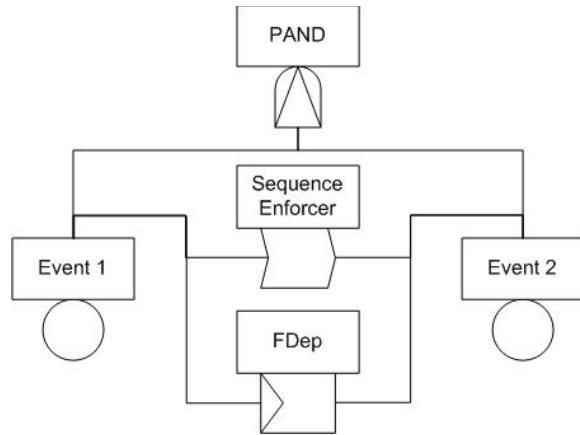


Figure 6. The DFT that revealed an error in Nova

7. DISCUSSION

Bounded exhaustive testing has advantages that suggest that it would be a good addition to the quality assurance toolkit. The sheer volume of tests gives it a bug-detecting ability that appears greater than that of manual, ad hoc testing, in which a suite comprises at most a few hundred tests. The inclusion of every test within the given scope ensures that most, if not all, bugs related to the handling of structures within the scope will be detected. And yet, unlike traditional approaches to using formal methods, and in light of Coppit and Sullivan’s findings on the cost effectiveness of selective formalization, bounded exhaustive testing appears to be within the budget parameters of everyday development projects.

At the same time, the approach is not a panacea. Its soundness may be compromised by errors in the oracle or in the specification from which tests are generated. So long as the specification is too weak, and the oracle errs on the side of incorrectly rejecting results, the consequence will be false alarms. If there are many of these, the approach will be impractical, but if there are relatively few, the root cause should be easy to diagnose. On the other hand, if the specification is too strong (so that too few tests are generated), or the oracle incorrectly accepts results it should reject, the approach will fail to find errors that should have been detected. Egregious problems of this sort are likely to be detected. If the specification is far too constraining, the size of the test suite will obviously be too small. And to guard against an uncritical oracle, the oracle might itself be evaluated by a kind of mutation testing (injecting bugs into the program under test and checking that the oracle finds them).

A fundamental limitation of the approach shared with all testing methods is its incompleteness. It is always possible, in general, that a behavior just beyond the tested bound will be erroneous. A more subtle problem shared with all systematic testing methods is that one cannot draw any conclusions about the statistical reliability of a system on the basis of bounded exhaustive input coverage.

Nevertheless, coverage metrics are widely used to build confidence in software, and the criterion underlying BET—coverage of inputs data up to certain sizes—should be no exception. It might also be possible to collect data that correlates the criterion with known remaining bug densities. The more specific the data to the kind of program, the more credible the statistical data will be. In particular, data collected for one version of a program, or for members of a program family, should be very relevant.

There is evidence [33] from the analysis of the Java library—smaller but nevertheless industrially relevant programs—that input space coverage correlates strongly with code coverage. As the bound on data structure size was increased, code coverage increased in tandem, very rapidly. Moreover, code coverage shouldered off at small bounds (structures containing 5 or 6 nodes), achieving a higher level of coverage than that obtained by a random test suite of the same size (containing both large and small inputs). This lends further credibility to our approach, assuming that the correlation between code coverage and fault detection can be demonstrated.

Less fundamentally, but still significant, are the sources of incompleteness arising from the limited focus of the testing effort. Reverse engineering from the existing system to a specification is by necessity selective; test cases are generated only for some components of the system, and only for some aspects of their behavior. For example, we tested the code that translates fault trees in abstract syntax to BDD’s and Markov chains, and the code for solving these forms, but not the compilation code that generated the abstract trees from the their concrete syntax. We also omitted certain dynamic fault tree constructs from our specification, such as *phase-or* gates, which are used in modeling phased-mission systems. This kind of incompleteness is not specific to our approach; it is inevitable in the cost/benefit tradeoff in deciding how to target testing.

TestEra, our underlying test generation mechanism, does not handle numeric data types well. Being based on Alloy, a first order logic of relations, it handles the complexities of data structures well, but has only minimal support for integers and none for real numbers. This, among other things, means that scalar parameters of fault trees (such as probabilistic failure rates represented by floating point numbers) cannot be covered. It may be possible to incorporate an additional constraint solver to handle numeric components, in the same way that many theorem provers now combine numeric and logical decision procedures.

Finally, since the test cases were generated from an Alloy specification distinct from the original Z specification, there was a further risk of compromising completeness. And indeed, our first version of the Alloy specification did fail to generate all legal trees. Alloy’s visualization facility, in which a series of generated trees can be examined on-screen, helps mitigate this problem. It does suggest however that ideally there should be no additional translation step, and a single specification should suffice.

Despite the clear limitations of bounded exhaustive testing, it is clear that it has already had a positive cost/benefit outcome for the Galileo project. We believe it to be a good contender for future projects, and are guardedly confident that improvements in the technique will have the potential for industrial impact.

8. RELATED WORK

In this section we survey related work on specification-based test generation and test selection criteria.

8.1. Evaluating Bounded Exhaustive Testing

A recent study [33] compares, for a variety of data structure implementations, bounded exhaustive testing with testing using randomly selected inputs using the Korat framework. The inputs in the random sample are within a larger bound on the input size but the number of inputs is fixed to be the same as the number in the exhaustive sample. For comparing suites, the criterion used is

mutation testing. The results show that for the benchmark structures, bounded exhaustive testing outperforms random selection in a majority of the cases. It is worth pointing out that the inputs in the random sample were also originally generated with Korat; indeed, inputs with complex structure cannot feasibly be generated in an equally-likely random fashion.

8.2. Specification-Based Test Generation

Korat [3] is a testing tool similar to TestEra [32], the tool we used. Like TestEra, it can exhaustively generate all non-isomorphic instances of structurally complex data structures (e.g. binary tree, linked list) up to certain size bounds (e.g. the number of nodes). But unlike TestEra, it takes as input constraints written as Java predicates. This has obvious merit in the unit testing of code modules, since programmers do not have to learn a new notation. But for our application, TestEra is more appropriate. Korat is very sensitive to the way in which the input constraint is written, since its generation algorithm follows the structure of the constraint. TestEra, because it employs the Alloy Analyzer’s translation to SAT, is largely insensitive to the constraint’s logical structure. This makes it easier to write the constraint, and to structure it as a conjunction of separate properties. For this kind of work, Alloy is anyway better suited to the description of the data structures than Java, since its relational operators allow a more succinct and abstract description.

Dick and Faivre [12] pioneered the idea of generating test cases automatically from model-based formal specifications. They developed the now well-known DNF approach. Later Helke et al [21] developed a technique based on the DNF approach for automatically generating test cases from Z specifications. They employed a theorem prover to support the generation and evaluated their approach by generating test cases from a steam boiler’s specification. Horcher [22] developed a technique for deriving test cases from a Z specification. Offut et al. [35] developed a technique to automatically generate test cases based on UML state-charts. They evaluated their approach on a system with only 400 lines of C and 7 functions. Chang et al. [5] developed Structural Specification-Based Testing (SST) using ADL as the formal specification language. Stocks et al. [41] developed the Test Template framework. They applied it to test the implementation of a symbol table, and a very small topological sort program [31]. Without any tool support, users had to manually create test cases.

Our work differs from all of these in two respects. First, the above techniques were designed to generate test cases for control-intensive systems; is able to generate complex structures. Second, the part of Galileo on which we have evaluated our approach is much larger than all the systems in previous case studies.

8.3. Other Test Generation Approaches

Automatic test case generation is of course an old idea, and there is a large literature on the topic. It is worthwhile to compare our approach with at least one related approach to give a sense of the differences. We compare our approach with that of Fisher, et al. for spreadsheet testing [17].

First, to generate test cases, Fisher et al. adapted the definition-usage-pairs (*du-pairs*) dataflow test adequacy criterion for imperative programs. Our work is based on an entirely different criterion: bounded exhaustive testing to computationally feasible bounds. Second, they adopted Ferguson and Korel’s *chaining approach* for generating test cases satisfying the *du-pairs* criterion.

Our approach is specification-based not implementation-based. Third, the inputs they generate are numerical vectors; ours are complex structures. Above all, their technique generates vectors of numbers based on the implementation of a spreadsheet to satisfy *du-pairs* coverage. Ours exhaustively generates complex structures up to a computationally feasible bound based on an abstract, formal specification.

8.4. Test Selection Criteria

Our criterion of exhausting a bounded *input space* differs from traditional testing criteria, such as statement and branch coverage, dataflow coverage [36], and modified condition/decision coverage [6]. Code-based model checkers, such as Java Pathfinder [20] and Verisoft [18], have traditionally focused on checking control intensive properties and not properties of data structures. A recent framework [27] (implemented using Java Pathfinder) shows how traditional symbolic execution [28] can be generalized to enable software model checkers for correctness checking and (non-isomorphic) test generation for (multi-threaded) programs that manipulate structurally complex data.

To the best of our knowledge, our work is the first to evaluate the effectiveness of bounded exhaustive testing on a real large-scale system.

9. CONCLUSIONS AND FUTURE WORK

We have presented a technique for improving one’s confidence in software that brings together ideas from testing and automated formal methods. It benefits from the advantages of each: like testing, it can be applied *ex post facto*, and is insensitive to the code size; and like formal methods, it is capable of exposing bugs that have eluded other analyses. The key elements of the technique—selectively reverse engineering a specification from which both a characterization of well-formed inputs and an oracle are derived, and the automatic generation of a huge test suite that covers all inputs up to a given size—are not radical, but together form a potent combination. Our experience shows that the approach is feasible and effective, having revealed previously unknown flaws in a component that is being transitioned into production use.

If this experience can be consolidated in further experiments, the technique might become a useful tool in the certification toolkit. Exhausting all inputs up to a given size is intuitively appealing, and in practice reveals subtle errors (partly due simply to the size and density of the test suite). But it does not offer guarantees. We cannot conclude from a successful analysis that errors of a particular class are now absent, or that the probability of failure has been reduced by some known amount. Nor, in our experience, were we able to perform bounded exhaustive testing to a point that would convince us we had found all significant failure modes—say to trees of fifteen events, where complex interactions among fault tree language constructs might emerge. Clearly finding a way to obtain precise measures of increased dependability is a vital area of future work. In the meantime, however, the technique still offers a useful standard. The approach presented here appears particularly well suited to testing software systems exhibiting behaviorally simple but computationally complex processing of structurally complex inputs. Bounded exhaustive testing might thus be added to the traditional list of coverage criteria as another criterion that, while still giving no absolute assurances, at least allows one to recognize objectively a level of scrutiny that can be compared across projects, and for which statistical data can be collected.

10. ACKNOWLEDGMENTS

The work of Kevin Sullivan was supported in part by an ITR grant from the National Science Foundation (number 0086003). Daniel Jackson and Sarfraz Khurshid acknowledge support from the ITR program of the National Science Foundation (number 0086154), and from the NASA Ames High Dependability Computing Program (cooperative agreement NCC-2-1298). We thank David Evans for commenting on a version of this paper, Wei Le for helping with some experiments, Matthew Moskewicz for help with the mchaff SAT solver and for pointing out that mchaff is not optimized for enumeration. We thank Joanne Bechta Dugan and her students for serving as tireless and effective domain experts.

11. REFERENCES

- [1] Bayardo, R. J. Jr. and Schrag R. C. Using CSP look-back techniques to solve real-world SAT instances. In Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97), pages 203-208, Menlo Park, July 27-31 1997. AAAI Press.
- [2] Beizer, B. *Software Testing Techniques*. 2nd edition, Van Nostrand Reinhold, New York, USA, 1990.
- [3] Boyapati, C., Khurshid, S., and Marinov, D. Korat: Automated Testing Based on Java Predicates. *ISSTA'02*
- [4] Boyd, M. A. *Dynamic Fault Tree Models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems*. PhD thesis, Duke University, Department of Computer Science, Apr. 1991.
- [5] Chang, J., and Richardson, D. J. Structural specification-based testing: automated support and experimental evaluation. Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering
- [6] Chilenski, J. J., and Miller, S. P. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):191-200
- [7] Coppit, D, and Painter, R. Shared Semantic Domains for Computational Reliability Engineering. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 168-180, Denver, Colorado, 17-20 November 2003. IEEE.
- [8] Coppit, D., and Sullivan, K. J. Galileo: A tool built from mass-market applications. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 750-3, Limerick, Ireland, 4-11 June 2000. IEEE.
- [9] Coppit, D. *Engineering Modeling and Analysis: Sound Methods and Effective Tools*. PhD thesis, The University of Virginia, Charlottesville, Virginia, Jan. 2003. URL: <http://www.cs.wm.edu/~coppit/papers/dissertation.pdf>.
- [10] Coppit, D., and Sullivan, K. J. Sound methods and effective tools for engineering modeling and analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 198-207, Portland, Oregon, 3-10 May 2003. IEEE.
- [11] Coppit, D., Sullivan, K. J., and Dugan, J. B. Formal semantics of models for computational engineering: A case study on dynamic fault trees. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 270-282, San Jose, California, 8-11 Oct. 2000. IEEE.
- [12] Dick, J., and Faivre, A. Automating the generation and sequencing of test cases from model-based specifications. *FME'93*
- [13] Doyle, S. A., and Dugan, J. B. Dependability assessment using binary decision diagrams (bdd's). In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, pages 249-258, Pasadena, California, 27-30 July 1995.
- [14] Dugan, J. B., Bavuso, S., and Boyd, M. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363-77, Sept. 1992.
- [15] Dugan, J. B., Sullivan, K. J., and Coppit, D. Developing a low-cost high-quality software tool for dynamic fault tree analysis. In *IEEE Transactions on Reliability*, December 1999
- [16] Dugan, J. B., Venkataraman, B., and Gulati, R. DIFTree: A software package for the analysis of dynamic fault tree models. In *Annual Reliability and Maintainability Symposium 1997 Proceedings*, Philadelphia, PA, January 1997.
- [17] Fisher, M., Cao, M., Rothermel, G., Cook, C. R., Burnett, M. M. Automated Test Case Generation for Spreadsheets, *Proceedings of the 24th International Conference on Software Engineering*, May 2002, pages 241-251.
- [18] Godefroid, P. Model Checking for programming languages using VeriSoft. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 174-186, 1997.
- [19] Harrold, M. J. Testing: a roadmap. In *ICSE - Future of SE Track*, pages 61-72, 2000
- [20] Havelund, K. and Pressburger, T. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.
- [21] Helke, S., Neustupny, T., and Santen, T. Automating test case generation from Z specifications with Isabelle. *Lecture Notes in Computer Science*, 1212:52-71, 1997.
- [22] Horcher, H.-M. Improving software tests using Z specifications. In *Z User Meeting (ZUM'95)*, volume 967 of LNCS.
- [23] Jackson, D. Micromodels of software: Modelling and analysis with Alloy. 2001. <http://sdg.lcs.mit.edu/alloy/reference-manual.pdf>
- [24] Jackson, D., Schechter, I., and Shlyakhter, I. Alcoa: the Alloy Constraint Analyzer. In *Proc. International Conference on Software Engineering*. 2000. Limerick, Ireland.
- [25] Khurshid, S., Marinov, D., Shlyakhter, I., Jackson, D. A Case for Efficient Solution Enumeration. *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, S. Margherita Ligure - Portofino (Italy), May 2003.

- [26] Khurshid, S., and Marinov, D. “Checking a Java implementation of a naming architecture using TestEra,” In *Post-CAV Workshop on Software Model Checking*, volume 55(3) of *Electronic Notes in Theoretical Computer Science* (ENTCS), Paris, France, July 2001. Elsevier Science.
- [27] Khurshid, S., and Pasareanu, C., “Generalized symbolic execution for model checking and testing,” 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Warsaw, Poland, April, 2003.
- [28] King, J.C., “Symbolic execution and program testing,” *Communications of the ACM*, 19,7, 1976, pp. 385 – 394.
- [29] Knight, J. C. and N. G. Leveson, An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming, *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 96-109, January 1986.
- [30] Lions, J.L., Ariane 5: Flight 501 Failure, Report by the Inquiry Board, Paris, 1996.
- [31] MacColl, I., Carrington, D., and Stocks, P. An Experiment in Specification-based Testing. *Technical Report No. 96-05, Software Verification Research Centre*, Department of Computer Science, The University of Queensland. May 1996.
- [32] Marinov, D., and Khurshid, S. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.
- [33] Marinov, D., A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, “An evaluation of exhaustive testing for data structures,” MIT Computer Science and Artificial Intelligence Laboratory Report MIT-LCS-TR-921, September, 2003.
- [34] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
- [35] Offutt, J., and Abdurazik, A. Generating Test from UML Specifications. In *UML conference proceedings*, Fort Collins, CO, October 1999.
- [36] Rapps, S., and Weyuker, E. Selecting Software Test Data Using Data Flow Information. *IEEE Trans. on Software Eng.*, vol. SE-11, No. 4, Apr. 1985
- [37] Saaltink, M. The Z/EVES system. In J.P. Bowen, M. G. Hinchey, and D. Till, editors, ZUM'97: the Z Formal Specification Notation, Dth International Conference of Z Users, volume 1212 of *Lecture Notes in Computer Science*, pages 72-85. Springer-Verlag, 1997.
- [38] Schwartz, E. *Design and implementation of intentional names*. Master's thesis, MIT Laboratory for Computer Science, Cambridge, MA, June 1999.
- [39] Shlyakhter, L. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*. June 2001.
- [40] Spivey, J. M. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, Prentice-Hall, 1992.
- [41] Stocks, P., and Carrington, D. A Framework for Specification-Based Testing. *IEEE Trans. Software Eng.*, vol. 22, no. 11, pp. 777–793, 1996.
- [42] Sullivan, K. J., Dugan, J. B., and Coppit, D. The Galileo fault tree analysis tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 232-5, Madison, Wisconsin, 15-18 June 1999.
- [43] Sun Microsystems. Java 2 Platform, Standard Edition, v1.3.1 API Specification. <http://java.sun.com/j2se/1.3/docs/api/>
- [44] Vesely, W. E., Goldberg, F. F., Roberts, N. H., and Haasl, D. F. *Fault Tree Handbook*. U. S. Nuclear Regulatory Commission, NUREG-0492, Washington DC, 1981.
- [45] Weyuker, E. J., and Ostrand, T. J. Theories of program testing and the the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236-246, May 1980.