Software Assurance by Bounded Exhaustive Testing

David Coppit, *Member*, *IEEE*, Jinlin Yang,
Sarfraz Khurshid, Wei Le, and Kevin Sullivan, *Senior Member*, *IEEE*

Abstract—Bounded exhaustive testing (BET) is a verification technique in which software is automatically tested for all valid inputs up to specified size bounds. A particularly interesting case of BET arises in the context of systems that take structurally complex inputs. Early research suggests that the BET approach can reveal faults in small systems with inputs of low structural complexity, but its potential utility for larger systems with more complex input structures remains unclear. We set out to test its utility on one such system. We used Alloy and TestEra to generate inputs to test the Galileo dynamic fault tree analysis tool, for which we already had both a formal specification of the input space and a test oracle. An initial attempt to generate inputs using a straightforward translation of our specification to Alloy did not work well. The generator failed to generate inputs to meaningful bounds. We developed an approach in which we factored the specification, used TestEra to generate abstract inputs based on one factor, and passed the results through a postprocessor that reincorporated information from the second factor. Using this technique, we were able to generate test inputs to meaningful bounds, and the inputs revealed nontrivial faults in the Galileo implementation, our specification, and our oracle. Our results suggest that BET, combined with specification abstraction and factoring techniques, could become a valuable addition to our verification toolkit and that further investigation is warranted.

Index Terms—Formal methods, program verification, testing and debugging.

1 Introduction

Assuring the trustworthiness of even modestly complex software remains a daunting and important problem. Exhaustively testing software for all possible inputs is effective in theory, but usually infeasible owing to the vast sizes of many input spaces [1]. Instead, researchers have developed a number of test selection criteria to identify suitable sets of test cases [3], [7], [34], [48].

Recent research has demonstrated the feasibility of employing *bounded exhaustive testing* (BET) to test small software modules, such as programs that manipulate simple data structures [32]. The underlying rationale for BET lies in Jackson's *small scope hypothesis* [24]: that any given fault is likely to be revealed by testing some small input and that testing all inputs up to some small scope thus suffices to reveal a high proportion of faults. Despite promising early results and the continuing advance of hardware capability, we do not yet know if BET can be feasibly and usefully applied to larger systems taking more complex inputs.

BET can always be used within trivial bounds. The question is whether it can produce enough inputs, and whether those inputs can be run, to meaningfully test meaningful systems. The main contribution of this paper is a limited positive finding, based on one experiment, that BET, perhaps combined with specification abstraction and decomposition techniques, has the potential to produce meaningful test results.

The subject of our experimental evaluation was Galileo [10], [16], [43], a tool for modeling and analyzing the reliability of complex, fault-tolerant, computer-based systems such as aircraft. Galileo analyzes models in the form of dynamic fault trees (DFTs) [14], [15], [46] to estimate system reliability. DFTs constitute a significantly more complex input space than has been explored in prior studies of BET.

Our results rest on two separate efforts. First, we leveraged our previous work on the formal specification and validation of the syntax and semantics of DFTs [8], [11], and the production of a relatively verifiable analyzer that sacrificed performance scalability for ease of verification. That analyzer served as our test oracle. Second—the new effort—we automated bounded exhaustive generation of DFT inputs by translating our specification into Alloy [23], and using TestEra [32] as a generator.

Our initial attempts failed. Alloy would run out of memory long before it generated a reasonable space of input structures. We eventually overcame the problem by decomposing and abstracting the input specification in a way that allowed us to offload work from Alloy. We abstracted certain details from the specification, recorded the details separately, used TestEra to generate abstract inputs from the abstracted specification, and finally recombined the abstract inputs with the details to reconstitute the input space.

Using this technique, we were able to generate input structures to meaningful bounds—to the point that testing

- *D. Coppit is with the Department of Computer Science, The College of William and Mary, Williamsburg, VA 23187. E-mail: david@coppit.org.*
- *J. Yang, W. Le, and K. Sullivan are with the Department of Computer Science, The University of Virginia, Charlottesville, VA 22901. E-mail: {jy6q, weile, sullivan}@virginia.edu.*
- *S. Khurshid is with Electrical and Computer Engineering, University of Texas, Austin, TX 78712. E-mail: khurshid@ece.utexas.edu.*

Manuscript received 26 Oct. 2004; revised 9 Apr. 2005; accepted 13 Apr. 2005; published online 26 May 2005.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0248-1004.

did reveal some significant faults and to the point where running the test cases, rather than generating them, became the bottleneck. We generated and tested more than 250,000 dynamic fault tree inputs.

We have structured the rest of this paper as follows: The next section introduces dynamic fault tree analysis and the Galileo tool. Section 3 describes the setup of our case study, focusing on the input generation and output validation aspects of the testing process. Section 4 discusses the key limiting factors we discovered during preliminary experiments and presents our solution to the input generation problem. Section 5 presents the results of our case study, which we discuss in Section 6. Section 7 presents related work. Section 8 concludes.

2 Dynamic Fault Trees and Galileo

In this section, we introduce dynamic fault tree analysis and the subject of our study, the Galileo dynamic fault tree modeling and analysis tool. Galileo allows reliability engineers to estimate the reliability of complex systems using the DFT method. Galileo is in limited production use at NASA now, and is one of two major new probabilistic risk assessment (PRA) methods and tools planned for agency-wide adoption.

2.1 Dynamic Fault Tree Analysis

Dynamic fault tree modeling and analysis [5], [15], [46] allows reliability engineers to estimate reliability and other dependability properties of computer-based systems using complex redundancy management. Such systems are essential in space, military, nuclear, and other critical, hazardous applications. The Galileo DFT notation supports coverage modeling, order-dependent failure, replicated events, phased missions, common cause and cascading failures, shared spares, numerous time-to-failure distributions, and more.

A dynamic fault tree (DFT) is a graph in which *basic event* nodes represent component failure events or other events that bear on system functioning, and in which *gate* nodes represent events that occur as functions of other events. Gates can be either static or dynamic, depending on whether it matters in what order their input events occur. For example, a static *AND gate* models a failure event that occurs if and only if all of its input failure events occur. A dynamic *priority-AND gate* (PAND) fails if and only if all of its input gates fail *in order*. We say that a node *fails* as a shorthand meaning that the event that the node models has *occurred*. *Constraint* nodes assert constraints on events. A *functional dependency* constraint (FDEP) models a cascading failure by stating that one event triggers immediate, simultaneous occurence of other basic events. A *sequence enforcing* constraint (SEQ) states that given failures can occur only in a given order.

Fig. 1 presents an example dynamic fault tree with a spare gate, three basic events as inputs to the spare gate, and a functional dependency constraint connecting the first and third basic events. A spare gate fails if and only if all of its inputs—which model redundant parts—have either failed or become unavailable. Parts can become unavailable

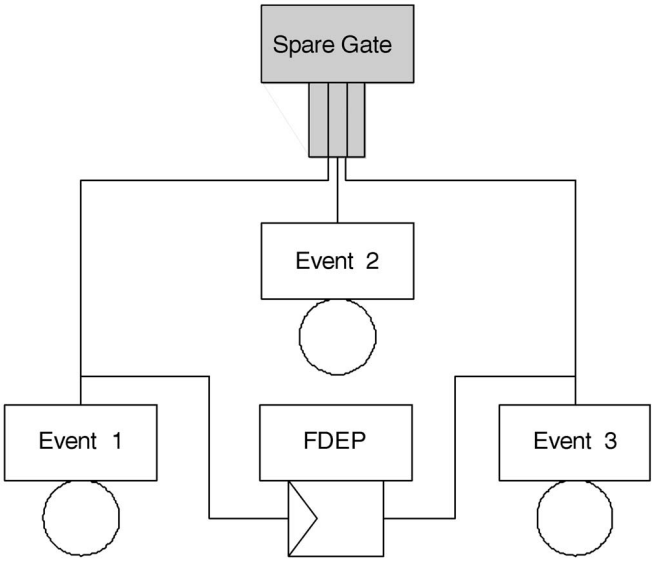


Fig. 1. A dynamic fault tree involving a spare gate, three basic events, and a functional dependence (cascading failure) constraint.

in systems with shared spares. Operationally, a spare gate functions for as long as a working part is available, and it fails when there are no more working available parts. It uses parts in the order in which they appear as inputs. Our spare gate thus starts in a state in which *Event 1* is in use and, thus, unavailable to any other spare gate. If *Event 1* fails, the spare gate attempts to allocate *Event 2*. The functional dependence constaint indicates that the failure of *Event 1* causes the failure of *Event 3*. In this system, then, having *Event 1* and *Event 2* fail suffices to cause the spare gate as a whole to fail.

In practice, time-to-failure distributions are associated with basic events, and time-to-failure distributions are computed for the top-level node of a fault tree, which models an overall system failure. Fault tree analysis is used to estimate numerous properties of systems being modeled. For example, given a fault tree, failure rates for basic events, and an expected mission time, one can compute system reliability. Such analyses are performed by converting fault trees into state-based or combinatorial representations [5]. In the state-based approach, for example, one converts the fault tree into a continuous-time Markov chain where each state represents an equivalence class of failure histories, and each transition corresponds to the failure of a component. In some histories, the top event in a tree has occurred. One computes the probability of failure by solving the Markov chain and determining the probability of being in any state in which the top-level event has failed.

Fig. 2 shows a state machine for the example tree. 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 failure state, as indicated by the failed status of the (top-level) spare gate.

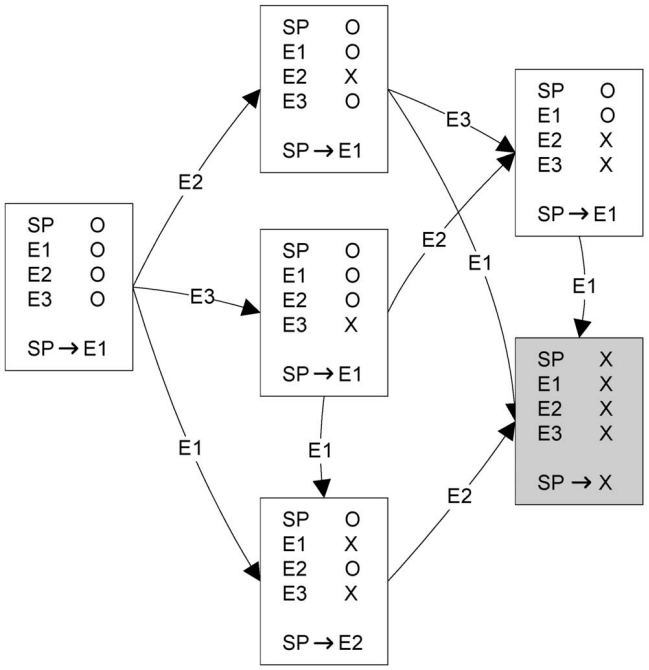


Fig. 2. The state space for the DFT in Fig. 1.

Our example hints at the difficulties one faces in developing a complex new modeling method and software implementation to automate it. One challenge is to specify and validate the modeling syntax. We spent a considerable amount of time debating whether certain DFTs made sense: whether they were trees that could conceivably arise in actual modeling practice. Second, it is difficult to specify a complete, precise, abstract semantics for the space of all syntactically valid models. Although the semantics of any given gate (such as PAND) are easy to describe, it is not easy to discern the meaning of dynamic fault trees in which many such constructs interact. For instance, functional dependencies cause simultaneous failures, and PAND gates fail if their inputs occur in order. Do simultaneous input events count? We had to resolve many such issues in developing an adequate semantics. Third, it is difficult to develop an efficient and dependable implementation of the specified mapping from fault tree expressions to analysis results, given that complex optimizations are needed to mitigate combinatorial explosion in expanding partial failure states. Building confidence in the dependability of a complex, optimized implementation is a challenge. There is clearly no silver bullet. In this paper, we simply ask can BET help?

2.2 Galileo

Galileo [10], [11], [16], [43] is a tool for modeling and analyzing DFTs. It began as a research prototype for assessing the viability of using mass-market applications as components. The success of the approach and prototype tool led NASA Langley Research Center to support us in developing the work, and they have now deployed it for use within the agency, e.g., on the International Space Station project.

These developments have placed a greater premium on the dependability of the core analysis functions. New modeling and analysis 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 is of increasing concern.

The Galileo DFT solver is an interesting case study for several reasons. First, DFTs are much more structurally complex as inputs than the red-black trees and such data structures used in previous evaluations of BET. Second, the program is larger and more complex than systems previously used in evaluating BET. The translation of dynamic fault trees to complex structures involving Markov chains and other representations is not simple.

These routines are complicated by optimizations addressing the combinatorial explosion problem. First, Galileo decomposes large DFTs into independent subtrees when it can, solves the subtrees independently, and composes the intermediate results to produce the final result [18]. Second, Galileo automatically chooses the faster of two DFT solvers: a *dynamic solver* that translates to Markov chains and a *static solver* that translates to BDDs [14]. The dynamic solver can solve most trees, and is the focus of our case study. The static solver is applicable only to trees without dynamic gates. For such cases, however, it can be much more efficient. A key required and expected property is that both solvers produce exactly the same answers for inputs for which they are both applicable. As we describe in Section 3.1, we utilized the static solver whenever possible to serve as an additional oracle for validating the results of the dynamic solver.

3 Case Study Design

We designed our study to evaluate the feasibility of using BET for meaningful testing of systems such as Galileo. Our feasibility criterion had three parts. First, BET should generate test inputs to bounds that we knew to include certain known fault-revealing inputs for Galileo. Second, BET should generate test inputs to bounds at which unexpected interactions among modeling constructs are possible—because that is where we believe faults are likely to cluster. Third, BET should reveal previously unknown faults.

The size of a structurally complex input is multidimensional. For example, a DFT could be measured in number of events, number of constraints, or number of connections. To control the overall size of the inputs that BET generates, one must set individual scope values for each such dimension. The three dimensions in our experiment were the number of events (basic events plus gates), the number of functional dependence constraints, and the number of sequence enforcing constraints.

As part of our evaluation, we iteratively increased these three scope values in an attempt to include fault trees that exercise known subtleties in the DFT semantics. Fig. 3 shows two such fault trees. The fault tree on the left illustrates "lack of replicate identity." Here, the *3* in the circle under *Event 3* indicates that *Event 3* is a *replicated event*—a node that models three functionally and statistically identical, anonymous components. Replication is a notational convenience that allows a reliability engineer to

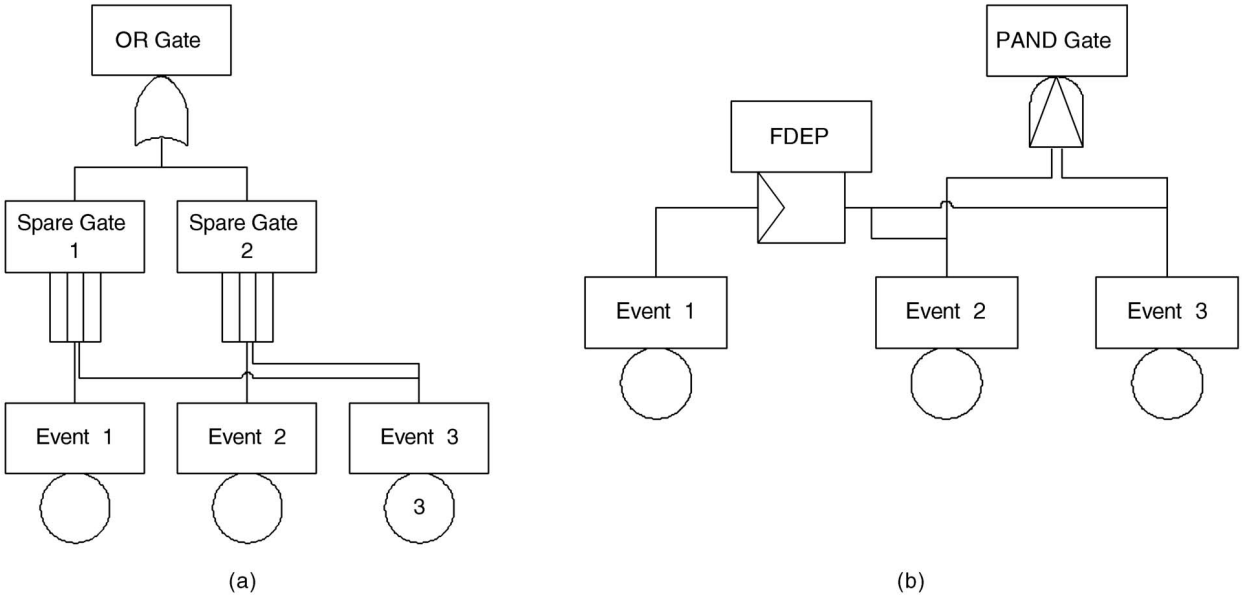


Fig. 3. Two fault trees with subtle semantics.

use a single basic event to model multiple components. In the state in which *Event 1* and *Event 2* have failed, both spare gates are using replicates of *Event 3*. When one of the three replicates fails, it is not clear whether it's the replicate in use by *Spare Gate 1*, *Spare Gate 2*, or neither.

The second fault tree illustrates a "simultaneous failure conflict." In this case, the functional dependence states that the two dependent basic events *Event 2* and *Event 3* fail at the same time that the trigger *Event 1* fails. The semantics of the priority and gate were not clear in the case where *Event 2* and *Event 3* fail at the same time.

The fault tree in Fig. 3a contains six events. The fault tree in Fig. 3b contains four events and one functional dependency. Based on these fault trees, we set our scope targets to be six gates or basic events, one functional dependence constraint, and one sequence enforcing constraint.

3.1 Evaluation Methodology

We first prepared the Galileo system to support testing. In addition to a test driver, we required both automated input generation and automated output validation. To generate inputs exhaustively starting with the smallest, we used *TestEra* as a state-of-the-art input generation technology. We checked outputs using our previously developed *Nova solver* as a test oracle. Unlike Galileo, we had derived Nova by careful engineering from our previously developed formal specification, eschewing complex optimization for ease of verification. We discuss these aspects of the case study in more detail in the next two subsections.

Our process starts with the smallest scope values for various characteristics of fault trees: one event, zero FDEPs, and zero SEQs. We generate all DFTs at that scope, test the dynamic solver for the DFTs, then increment the scope values and repeat generation and testing. We stop when we reach our goal scope, or when generation or testing of the software becomes infeasible.

To test at a given scope, we present generated test inputs to the Galileo dynamic solver, to the Nova oracle, and, if possible, to the Galileo static solver. We capture and compare the outputs. On finding a discrepancy, we ask domain experts acting as human oracles to hand-compute the correct output. The smallness of inputs is advantageous here: It makes manual computation of correct answers relatively easy. Finally, we assess blame: to Galileo, to the oracle implementation, or to the formal specification.

If BET detects a fault at all, it often manifests itself in a large number of input cases. When there are several faults, the problem becomes one of partitioning large numbers of observed failures into small numbers of equivalence classes by faults. Our approach is to start by selecting one failing test case and then ask our domain experts to debug the problem. Once we understand the problem, we then either fix it right away or modify the generator input specification to cause the generator to mark each input known to exercise the fault. Modifying the test harness to skip marked tests in the next round in effect focuses our testing on the remaining unexplained faults. As this process unfolds, an interesting side effect is that the specification evolves from one that describes the desired system to one that describes the actual system, with faults included but marked as such. We iterate until no more faults are revealed, at which point we increase the scope and continue generation and testing.

3.2 Automating Input Generation: TestEra

In our initial experiments, TestEra was unable to generate fault trees beyond a scope of three events (e.g., two basic events and an *AND* gate). The problem was clearly in the size of the state space that we were asking TestEra to expand. The need was to reduce the state space without losing information because we did, in the end, want to generate all specified inputs within scope. Our solution was to develop a two-stage process that uses TestEra to generate abstract fault trees, described by an abstracted specification, and then to patch up these abstract inputs using a

postprocessor that reconstitutes concrete fault tree inputs from the abstract inputs and from an auxiliary data structure that describes the information that we had abstracted from the specification. We defer discussion of this work-around to the next section, focusing here on TestEra itself.

TestEra [32] is a framework for automated specification-based testing of Java classes. To test a method, one provides a specification that consists of a precondition that describes allowed inputs to a method and a postcondition that describes the expected outputs. TestEra uses the precondition to 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 postcondition 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 [25] finds instances of Alloy specifications using an off-the-shelf SAT solver that supports enumeration, such as mChaff [35] or relsat [2].

TestEra supports efficient testing of isomorphic inputs by identifying the subset of nonisomorphic inputs that form a *revealing subdomain* [47]. The analyzer has automatic symmetry breaking [40] to eliminate many isomorphic inputs. TestEra can also exploit user-provided total orders to guarantee generation of exactly nonisomorphic inputs [26]. Considering only nonisomorphic inputs reduces the time required to test the program, without reducing the possibility of detecting faults.

Initial case studies with TestEra focused on checking Java programs. TestEra exposed faults in a naming architecture for dynamic networks [26], and in part of an earlier version of the Alloy Analyzer [32]. TestEra was also used to check methods on Java data structures, such as those in the Java Collection Framework [45].

Although TestEra was designed to test Java programs, we were able to modify it to serve as an input generator for our case study. Instead of writing preconditions of class member functions in Alloy, we provided TestEra with an Alloy specification of DFTs directly. We also wrote Java and Perl programs to interface with the TestEra generation back-end, translating the Java object that it generated into a textual representation of the DFT that could be read by Galileo. In this way, we were able to leverage the input generation capabilities of TestEra to enumerate all instances that satisfied the input specification. By changing the scope values in the specification, we could control the size of the generated inputs.

3.3 Automating Output Validation: Nova Solver

To check test outputs, we used an implementation of the Galileo dynamic analyzer called *Nova* as an oracle. Nova is far simpler than the Galileo version because it eschews optimizations (and is thus largely useless for analyzing large inputs). We implemented Nova as a direct translation into C++ of our previously developed formal specification of the syntax and semantics of DFTs [8], [11], [12], [38], written in Z [41].

We have developed the specification and Nova itself as part of an overall effort to evaluate the feasibility of combining formal methods and package-oriented programming to achieve both dependability and usability in engineering tools at low cost [8]. In creating Nova, we traded efficiency for dependability. Although produced for other reasons, Nova turned out to provide a good, though not perfect, oracle: a relatively dependable implementation able at least to handle small inputs. For dynamic fault trees with the four nodes, Galileo is about five times as fast as the oracle. We thus exploited the nature of BET—that it involves only small inputs—to ease the problem of developing (in this case, finding) a suitably dependable oracle.

The Nova DFT analyzer differs from Galileo in several ways. First, it supports a slight variant of the Galileo DFT language that we designed for improved regularity and orthogonality. The slight difference in input language was an inconvenience at times. We worked around it, mainly by lightly preprocessing generated inputs before dispatching them to Galileo and Nova, respectively. Nova provides a DFT dynamic solver, a DFT data structure, and a textual DFT parser. It does not support many Galileo functions, including phased missions, sensitivity analysis, or other advanced capabilities. However, it does support the core reliability analysis function of Galileo, which is what we had set out to test.

4 Preliminary Study of Limiting Factors

Our initial attempts to generate inputs to test Galileo revealed two impediments to the use of BET. First and foremost, TestEra was unable to generate DFTs beyond a small bound of three events. This was due in part to the combinatorial explosion of DFTs even for small scopes, and in part to the use of mChaff as an underlying SAT solver, which its author described as optimized for instance-finding, not exhaustive generation. Nevertheless, TestEra is a state-of-the-art tool that had been used to generate simpler inputs, and we remained hopeful that it could be made to work for our more complex inputs, as well.

The second bottleneck that emerged over time was the performance of the oracle. Because we designed Nova for verifiability, not performance, it took no measures against exponential growth in the size of the state space to be processed, and its execution time grew hugely as the size of DFT inputs increased. Throughout most of our effort, however, generating the DFTs remained the primary bottleneck.

Given our preliminary results, we focused on improving the scalability of input generation. We used several strategies. Following Khurshid et al. [26], we added symmetry breaking predicates to the specification provided to TestEra, to avoid generating fault trees that were isomorphic to previously generated inputs. However, even with symmetry breaking, the input space was too large. Next, we tightened our specification to exclude fault trees accepted by Nova, but that we knew would be rejected by Galileo. (This is one place where the slight language differences were an inconvenience.) For instance, we

TABLE 1
The Number of DFTs and AFTs Generated

Events	Seqs	FDeps	AFTs	DFTs
3	0	0	12	64
3	0	1	19	76
3	1	0	24	128
3	1	1	38	152
4	0	0	49	1,737
4	0	1	202	4,062
4	1	0	588	20,884
4	1	1	2,424	48,744
5	0	0	354	192,039
5	0	1	1,212	637,575
5	1	0	14,048	7,978,228
5	1	1	132,120	38,254,500
6	0	0	5,844	94,104,158
7	0	0	259,474	Not measured
8	0	0	1,719,872	Not measured

restricted the generation of DFTs having disconnected components, which Nova accepts but Galileo rejects.

While these techniques did allow us to extend the generation bound, the most effective technique was what we called *separation of generation concerns*. This approach combines *specification abstraction* (the generation of inputs satisfying the abstracted specification) and *postprocessing* of the abstract inputs to restore the details initially abstracted away. This approach offloads relatively trivial state space expansion work from the constraint-solving generator to a simple postprocessor, significantly reducing the state space that the generator has to traverse. For example, scalar value attributes of basic events (such as parameters of time-to-failure distributions) do not influence DFT structures at all and need not be managed by a constraint-solver. Rather, a simple postprocessor can just fill in different concrete values in where an abstract value appears in a generated abstract input. In this way, the constraint solver's workload is lightened, but the same results are obtained in the end.

Our approach is thus to abstract details from the specification that need not be handled by the constraint-solving generator and to replace them with abstract placeholders. This approach allows the constraint solver to handle only the hard parts of structure generation, while simple dimensions of the input space that merely exacerbate the curse of dimensionality are handled by a post-processing phase. We called the inputs generated from our abstracted specification *abstract fault trees (AFTs)*. We used Perl scripts to generate *concrete fault trees (FTs)* from the abstract ones by systematically substituting concrete values for the placeholders. The reduction in the state space that this factoring approach achieved enabled us to generate concrete trees with up to six events and abstract trees with up to eight. (See Table 1.)

The main benefit of this approach is that the generator can generate inputs exhaustively to larger bounds because the number of states within given bounds *as seen by the constraint-solving generator* is much smaller. We also found it is easier to check for—and at one point we detected—errors in the generation function by calculating the number of generated inputs expected within a given bound and checking the actual number against it.

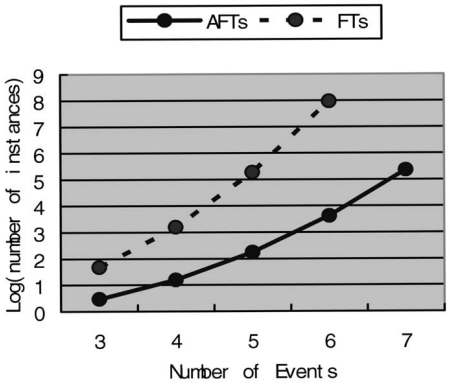


Fig. 4. Comparison of the number of AFTs and DFTs.

Fig. 4 compares the number of fault trees and abstract fault trees generated as a function of scope: in particular, in terms of the number of 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.

Prior uses of TestEra have involved data translations that used a similar form of abstraction, but they mapped each abstract input to one concrete input [33]. The bottom line result in our case was that we were able to transcend input generation as a bottleneck and increase the generation of test inputs to meaningful bounds.

5 Results

We now present the results of our application of BET to the Galileo solver.

5.1 Executing the Tests

For fault tree 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. After generating the inputs for a particular set of scope values, we ran the Galileo dynamic solver, Nova, and possibly the static solver for the inputs, and compared the results. The machine used to run test cases was a dual-CPU Pentium 4 at 3GHz with 1GB RAM and Windows XP Professional Service Pack 1.

Table 1 summarizes the number of inputs we generated. We were able to generate all abstract fault trees up to scopes of eight events and no functional dependencies or sequence enforcers. We were able to instantiate all concrete DFTs with up to six events and no constraints.

Having addressed the input generation limitation, the speed of the oracle became a more significant factor. Of the fault trees we generated, we were able to test all fault trees up to a scope of five events and no constraints. We tested approximately 250,000 inputs in this scope, taking about a week to complete. As expected, the performance of the oracle degraded as the size of the input fault tree increased. As a result, it would be impossible to run all trees with six

events without optimizing the oracle (which would be an option, in practice). The Nova solver takes 2 minutes to solve a six-event fault tree, so it would take 355 years to run all such DFTs.

5.2 A Fault in Galileo

BET 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. The example DFT presented earlier in Fig. 1 shows a test input that revealed a fault in Galileo. As discussed in Section 2, 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* or *Event 3* do. Consulting the state machine in Fig. 2, from the initial state in which no events have failed, a transition for the failure *Event 1* should lead to a state in which *Event 2* and the spare gate remain operational (the lower-most state in the figure).

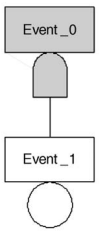
However, debugging of Galileo revealed that the transition led to a state with *Event 2* operational but not the spare gate. The resulting unreliability estimate for this model was thus higher than the correct value. This fault had gone undetected for at least three reasons. First, the DFT is one probably never previously considered, given the currently limited usage of the tool. It is just the kind of case that BET aims to test. Second, the fault is in the interaction of two novel constructs: functional dependencies and multiple spares. Third, the output is not obviously wrong, so it would not be caught easily during development.

5.3 A Fault in the Specification

One of the more interesting faults that our testing revealed was the omission of an important precondition in the specification. The fault was both revealed by N-version programming [30] and showed that we had been "bitten" by one of its known risks. Recall that we used the static DFT solver as a second oracle when applicable. We assumed that if the Nova oracle result agreed with one of the solvers, then there must be a fault in the other solver. That assumption turned out to be wrong.

One class of test inputs revealed a discrepancy. Galileo's dynamic solver translates fault trees to Markov chains. It agreed with the oracle, which also translates them to Markov chains (albeit usually much larger ones). Galileo's static solver, which translates fault trees to binary decision diagrams, disagreed with both. We concluded that there must be a fault in the static solver. We submitted one of the input fault trees to our domain experts to verify that the static solver did not implement its specification.

To our surprise, our colleagues reported that the static solver calculated the correct answer according to published accounts of BDD-based techniques. We then assumed that the dynamic solver and oracle must have been exhibiting an identical failure—a classic N-version programming problem. However, when we asked our experts to compute the solution by hand using Markov techniques, they reported that once again the manually computed solution matched the ones produced by the two Markov solvers. This odd result created much confusion, because everyone knew from the literature that the Markov and BBD techniques should produce exactly the same answers in cases where they're both applicable.



toplevel Event_0;

Event_0 and Event_1;

Event_1 **lambda=.01** cov=0 res=.5 repl= 2 dorm=.5;

Fig. 5. The DFT that revealed a specification error.

A weekend's inquiry by a leading domain expert resolved the apparent contradiction. 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. Fig. 5 shows the simple fault tree that first revealed this fault. *Event 1*, a basic event, has a large lambda value of 0.01 for its failure rate.

This condition had been forgotten, in a sense. Our testing rediscovered it. The Markov-based dynamic solvers (Galileo and Nova) agreed, but were both wrong because no answer should have been produced for the precondition-violating input DFT. We thus had, in some sense, an N-version programming failure, with the common faulty behavior traceable to a common faulty specification. The BDD-based solver had the same underlying fault—not checking a required precondition on the input—but the BDD algorithm misbehaved in a different way. We thus detected the fault because of algorithmic diversity, a small N-version programming success. Bounded exhaustive testing was able to quickly reveal this subtle fault.

5.4 A Fault in the Nova Test Oracle

We also found a fault in Nova's implementation of sequence enforcement constraints. Recall that a sequence enforcer precludes events occuring in other than a specified order. In Fig. 6, the sequence enforcer states that *Event 2* cannot fail before *Event 1*. The implementation should invalidate system states with out-of-order failures, but Nova erroneously omitted this check. Our informal derivation of the Nova oracle from our formal specifications did not eliminate programmer error in this case.

6 Discussion

We now discuss limitations in our study, benefits and limitations of BET, and the practical challenges of employing the approach.

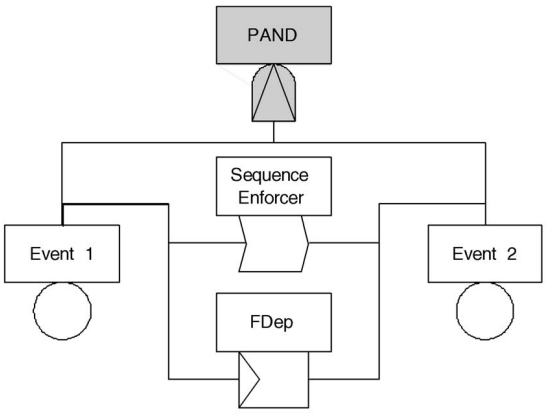


Fig. 6. The DFT that revealed a fault in Nova.

6.1 Limitations of the Case Study

There are a number of limitations in our study. The first is our choice of tools. For example, TestEra does not handle numeric data types well. Being based on Alloy, a first order logic of relations, it handles complex data structures, but has only minimal support for integers and none for real numbers. Thus, the tool cannot easily generate scalar parameters of fault trees, such as basic event failure rates represented by floating point numbers. We simply programmed our postprocessor to sample from a small set of possible scalar values. In cases where scalar features are subject to complex constraints, it would be necessary to use an additional constraint solver designed for this purpose.

Similarly, our choice to trade performance for verifiability in our oracle limited the feasible scope in our experiment. Moreover, as we saw, manually implementing the oracle created fault introduction risks. Using an executable specification as an oracle might have helped, had we had a suitably expressive notation, but it might have created even more serious problems in terms of oracle performance.

In terms of the inputs, recall that we had developed an Alloy specification of DFTs to be used by TestEra to generate instances. This Alloy specification was a translation of our Z specification. This translation step also introduced the possibility of error. 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 onscreen, can help mitigate this problem. It does suggest however that ideally there should be no additional translation step, and that a single specification should suffice. An executable specification that could serve as both an oracle and for test input generation would be particularly attractive.

Despite the limitations of our case study, BET did reveal subtle and substantial faults in the dynamic solver and its specification. Overall, our experiment constitutes a single test of BET on a single but interesting system. We achieved limited but positive results. We advise caution in inferring too much from our results. First, the level of testing we were able to perform did not leave us as convinced as we wanted to be of the correctness of Galileo across the whole input space. We could not generate DFTs to a large enough size to test all relevant combinations of modeling features. Even if we could generate them, we could not run them because their number is vast; moreover, our oracle, without optimizations, could not handle much larger inputs that we did run. Second, ours is just one experiment. Other systems will have different characteristics, and our results do not necessarily predict the results others would achieve. Third, we had to employ special measures to get the success we had—in the refactoring of the input specification. It is not clear how complex specifications will be for other applications, or whether similar opportunities would exist for state space reduction. Our results do suggest that BET has potential to provide a useful testing technique. Additional efforts and experiments to develop and evaluate its potential appear to be justified.

6.2 Strengths and Weaknesses of the Approach

Bounded exhaustive testing has properties that suggest that it might be a good addition to our quality assurance toolkit. The exhaustive coverage of an initial segment of the input space that it achieves give it a fault-detecting ability that in many cases seems likely to be greater than that of ad hoc testing, in which a suite comprises at most a few hundred, usually manually crafted, tests. The inclusion of every test within a given scope ensures that most, if not all, faults related to the handling of structures within the scope will be detected.

Our previous testing of Galileo included tests of all gates and constraints, tests involving boundary conditions in the input space, and some tests of cases involving subtle interactions among fault tree constructs. In some sense, we thus tested for suspected faults. BET, by contrast, like other coverage criteria, makes no assumptions about the distribution of fault-revealing inputs (save for Jackson's hypothesis). It is thus perhaps more likely to reveal unsuspected faults involving rare combinations of conditions. On the other hand, as with other coverage-based criteria, it is difficult at best to infer quantified improvements in fielded reliabity from the application of BET. It might be possible to collect data to correlate uses of BET with remaining fault densities—an idea that we leave to future work. The more specific such data are to given kinds of programs, the more credible they would be. In particular, data for one version of a program, or for members of a family, could be quite relevant.

There is also evidence [32] from the analysis of the Java library that input space coverage correlates strongly with code coverage. As the researchers increased the bounds on data structure size, code coverage increased rapidly, in tandem. Moreover, code coverage shouldered off at small bounds, achieving a higher level of coverage than that obtained by a random test suite of the same size containing both large and small inputs.

Like any testing technique, BET clearly has limitations. First, if there are faults in the oracle or specification, the approach is unsound. Second, the approach is also clearly incomplete. An adversary can always seed faults beyond any given scope. Third, as we have said, one cannot draw conclusions about the statistical reliability of a system based on the use of BET. Generated inputs generally do not represent a statistically valid sample of the inputs seen in production use. Fourth, performance bottlenecks limit the

use of the technique. We encountered two: first in the generator, then in the running of the test cases, especially in the use of our oracle to check Galileo outputs. Our specification factoring approach removed the generation bottleneck. Running test cases was then the problem. That bottleneck limited us still to testing within a relatively modest bound.

6.3 Achieving Effective Separation of Generation Concerns

The separation of generation concerns approach that we described in Section 4 turns out to be conceptually straightforward. However, it was only by making a mistake that we learned how to do it properly. We based our initial partitioning on a poor choice of abstraction boundary. The postprocessor ended up being more complex than necessary, which in turn led to our having introduced two faults. In the end, certain test cases that should have been generated were not.

The problem was that we overabstracted the gate type and, thereby, offloaded from Alloy not only trivial state space expansion tasks, but some constraint-satisfaction tasks, as well. We had abstracted AND, OR, PAND, THRESHOLD, and SPARE gates to a single abstract GATE type, with the idea that the postprocessor would simply fill in each abstract GATE token with one of the actual gate types. The problem was that spare gates are subject to connectivity constraints above and beyond those on AND, OR, PAND, and THRESHOLD gates. Overabstracting thus forced us to implement constraint reasoning for SPARE gates as a special case within the postprocessor. The programmer made a mistake in implementing this reasoning. First, he constrained the number of inputs to a spare gate to be no less than two. Second, he forgot that the order of the inputs to a spare gate affects the connectivity constraints on spare gates. The generator ended up not producing fault trees having spare gates with one input, and fault trees in which two spare gates share spares. These are important cases to test.

In studying the separation of generation concerns more closely, we observed that the key operation in our case study was to abstract distinctions between essentially equivalent values within certain domains in the specification. With respect to the constraints that define a DFT, AND and OR gates are interchangeable with each other, but not with SPARE gates. The latter are subject to additional constraints. The better factorization that we ended up with, abstracted AND, OR, and THRESHOLD to an abstract COMBINATORIAL gate. We recorded separately that COMBINATORIAL comprised AND, OR, and THRESHOLD. The postprocessor then reconstituted abstract inputs with COMBINATORIAL tokens by systematically substituting AND, OR, and THRESHOLD. Writing this sort of postprocessor is trivial. It could be automated quite easily. Automating the specification decomposition operation itself is another matter that we have just started to investigate. The question is: Can opportunities to abstract (in ways that keep the postprocessor free of constraint reasoning) be found without expanding the whole state space, which would defeat the purpose? In our experiment, we analyzed the specification and identified these opportunities by hand.

In addition to abstracting specifications and offloading the abstract elements to a postprocesser, it appears that *decomposing* (or *modularizing*) specifications might also be useful for enabling scalable test input generation. One could decompose a specification into nearly independent parts, generate partial inputs from the parts, and combine the parts in a postprocessing stage.

7 Related Work

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

7.1 Evaluating Bounded Exhaustive Testing

Previous work has demonstrated the feasibility and effectiveness of bounded exhaustive testing on 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], [39], with 2,000 lines of Java code. It was unclear that BET could scale to more complex systems This paper helps to fill that void with a study of the feasibility and potential utility of bounded exhaustive testing of a software system having more complex input structures and a code base at least five times larger.

A recent study [32] compares bounded exhaustive testing with testing using randomly selected inputs, for a variety of data structure implementations. The inputs in the random sample are within larger size bounds, but the number of inputs is the same as the number in the exhaustive sample. The research then compared the test sets using mutation testing. For the benchmark structures, bounded exhaustive testing outperformed random selection in a majority of the cases.

This paper is a revised and extended version of a previously published work [44]. Here, we describe our evaluation approach in more detail, describe the separation of generation concerns method, and present tests results corrected for the error in our first-version generator (as described above).

7.2 Specification-Based Test Generation

Korat [4] is a testing tool similar to TestEra [32]. Like TestEra, it can exhaustively generate all nonisomorphic instances of a given data structure type up to certain size bounds. Unlike TestEra, it takes input specifications written as Java predicates. This feature is useful in unit testing code modules because programmers do not have to learn a new notation. For applications such as ours, TestEra is more appropriate. Korat is highly sensitive to specification structure. TestEra uses the Alloy Analyzer's translation to SAT and is relatively insensitive to minor variations in the structure of the specification. It is easier to write useful input specifications for TestEra as conjunctions of separate properties. Alloy is also better suited than Java to describing data structures: Its relational operators allow for more succinct and abstract descriptions.

Dick and Faivre [13] pioneered the idea of generating test cases automatically from model-based formal specifications. They developed the 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 generation. They evaluated their approach by generating test cases from a steam boiler specification. Horcher [22] developed a technique for deriving test cases from a Z specification. Offut et al. [36] developed a technique to generate test cases based on UML state-charts. They evaluated their approach on a system with 400 lines of C and seven functions. Chang and Richardson [6] developed Structural Specification-Based Testing (SST) using ADL as the formal specification language. Stocks et al. [42] 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 tool support, users had to generate test cases by hand.

Our work differs from these in several respects. First, these techniques generate test cases for control-intensive systems; TestEra targets structurally complex inputs. Second, DFTs are structurally more complex than inputs spaces previously targeted, and the part of Galileo on which we evaluated our approach is larger than the systems in previous case studies.

7.3 Test Selection Criteria

Our criterion of exhausting a bounded input space differs from traditional adequacy criteria, such as statement and branch coverage, dataflow coverage [37], and modified condition/decision coverage [7]. Code-based model checkers, such as Java Pathfinder [20] and Verisoft [17], have traditionally focused on checking control properties. Recent work [28] based on Java PathFinder shows how traditional symbolic execution [29] can be generalized to enable software model checking. For example, the framework enables correctness checking and nonisomorphic input generation for programs that manipulate structurally complex data.

Harrold [19] points out that more research is needed on the effectiveness of test selection criteria in revealing faults. Our work is one step in evaluating the feasibility of bounded exhaustive testing on real systems. Assessing the effectiveness of the BET criterion relative to other criteria remains an important area of future work.

8 Conclusion

In this paper, we have tested the feasibility of applying bounded exhaustive testing to software systems taking structurally complex inputs. The basis for our evaluation was an experimental application of BET to the Galileo dynamic fault tree solver. The initial attempt failed, but factoring the specification and using an auxiliary post-processing step allowed us to test over 250,000 inputs. Testing within the feasible bounds revealed previously unknown faults in the implementation, specification, and in our test oracle.

This paper focuses primarily on the input generation bottleneck. Beyond the experiment itself, our core technical contribution is in the idea of factoring specifications in a principled way to reduce the load on a constraint solver while enabling the complete input space to be reconstituted by a simple postprocessor. Overall, we found the TestEra approach, augmented by our specification-factoring technique, profitable. We speculate that it is possible to automate both our specification factoring method and generation of the required postprocessors. Such a capability could provide a general-purpose optimization technique for constraint-based test input generation.

Our results suggest that BET could be a useful addition to our verification toolkit. Exhausting all inputs up to a given size is intuitively appealing and, in practice, can reveal subtle faults, but it is not a panacea. We cannot conclude from a successful test that faults of a particular class are absent, or that the probability of failure has been reduced by, or to, a known quantity. Nor, in our experience, were we able to perform bounded exhaustive testing to a point that would convince us we had found all significant faults. Our previous work had already revealed subtle modeling feature interactions in DFTs that arise only in cases involving 15 or more events. Clearly, finding ways to obtain precise measures of increased dependability is a vital area of future work. In the meantime, the technique still offers a useful standard. BET adds to the list of traditional coverage criteria, and as such provides an objective level of scrutiny that can be compared across projects, and for which statistical data can be collected.

Acknowledgments

The work of Kevin Sullivan was supported in part by an ITR grant from the US National Science Foundation (number 0086003). Daniel Jackson and Sarfraz Khurshid acknowledge support from the ITR program of the US National Science Foundation (number 0086154), and from the NASA Ames High Dependability Computing Program (cooperative agreement NCC-2-1298). The authors thank David Evans for commenting on a version of this paper and Matthew Moskewicz for helping them understand the performance of mChaff. They thank Joanne Bechta Dugan and her students for serving as domain experts. Finally, they thank Phyllis Frankl for discussions that helped them see the need to move from the original study to a generalized, albeit still not formal, statement of the underlying specification factoring approach.

References

[1] W.R. Adrion, M.A. Branstad, and J.C. Cherniavsky, "Validation, Verification, and Testing of Computer Software," *Comm. ACM,* vol. 14, no. 2, pp. 159-192, June 1982.

[2] R.J. Bayardo, Jr. and R.C. Schrag, "Using CSP Look-Back Techniques to Solve Real-World SAT Instances," *Proc. Nat'l Conf. Artificial Intelligence and Ann. Conf. Innovative Applications of Artificial Intelligence (AAAI/IAAI '97),* pp. 203-208, July 1997.

[3] B. Beizer, *Software Testing Techniques,* second ed. New York: Van Nostrand Reinhold, 1990.

[4] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '02),* pp. 203-208, July 2002.

[5] M.A. Boyd, "Dynamic Fault Tree Models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems," PhD dissertation, Dept. of Computer Science, Duke Univ., Durham, N.C., Apr. 1991.

[6] J. Chang and D.J. Richardson, "Structural Specification-Based Testing: Automated Support and Experimental Evaluation," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.*, Sept. 1999.

[7] J.J. Chilenski and S.P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Eng. J.*, vol. 9, no. 5, pp. 193-200, Sept. 1994.

[8] D. Coppit, "Engineering Modeling and Analysis: Sound Methods and Effective Tools," PhD dissertation, Dept. of Computer Science, The Univ. of Virginia, Charlottesville, Va., Jan. 2003.

[9] D. Coppit, R.R. Painter, and K.J. Sullivan, "Shared Semantic Domains for Computational Reliability Engineering," *Proc. Int'l Symp. Software Reliability Eng.*, pp. 168-180, Nov. 2003.

[10] D. Coppit and K.J. Sullivan, "Galileo: A Tool Built from Mass-Market Applications," *Proc. Int'l Conf. Software Eng.*, pp. 750-753, June 2000.

[11] D. Coppit and K.J. Sullivan, "Sound Methods and Effective Tools for Engineering Modeling and Analysis," *Proc. Int'l Conf. Software Eng.*, pp. 198-207, May 2003.

[12] D. Coppit, K.J. Sullivan, and J.B. Dugan, "Formal Semantics of Models for Computational Engineering: A Case Study on Dynamic Fault Trees," *Proc. Int'l Symp. Software Reliability Eng.*, pp. 270-282, Oct. 2000.

[13] J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications," *Proc. Conf. Formal Methods Eng.: Industrial-Strength Formal Methods*, pp. 268-284, Apr. 1993.

[14] S.A. Doyle and J.B. Dugan, "Dependability Assessment Using Binary Decision Diagrams (BDDs)," *Proc. Int'l Fault-Tolerant Computing Symp.*, pp. 249-258, July 1995.

[15] J.B. Dugan, S. Bavuso, and M. Boyd, "Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems," *IEEE Trans. Relativity*, vol. 41, no. 3, pp. 363-377, Sept. 1992.

[16] J.B. Dugan, K.J. Sullivan, and D. Coppit, "Developing a Low-Cost High-Quality Software Tool for Dynamic Fault Tree Analysis," *IEEE Trans. Relativity*, vol. 49, no. 1, pp. 49-59, Mar. 2000.

[17] P. Godefroid, "Model Checking for Programming Languages Using VeriSoft," *Proc. Conf. Principles of Programming Languages*, pp. 174-186, Jan. 1997.

[18] R. Gulati and J.B. Dugan, "A Modular Approach for Analyzing Static and Dynamic Fault Trees," *Proc. Ann. Reliability and Maintainability Conf.*, pp. 57-63, Jan. 1997.

[19] M.J. Harrold, "Testing: A Roadmap," *Proc. ICSE '00—Future of Software Eng.*, pp. 61-72, June 2000.

[20] K. Havelund and T. Pressburger, "Model Checking Java Programs Using Java PathFinder," *Int'l J. Software Tools for Technology Transfer*, vol. 2, no. 4, Apr. 2000.

[21] S. Helke, T. Neustupny, and T. Santen, "Automating Test Case Generation from Z Specifications with Isabelle," *LNCS*, vol. 1212, pp. 52-71, 1997.

[22] H.-M. Horcher, "Improving Software Tests Using Z Specifications," *LNCS*, vol. 967, pp. 152-166, 1995.

[23] D. Jackson, "Micromodels of Software: Modelling and Analysis with Alloy," http://sdg.lcs.mit.edu/alloy/reference-manual.pdf, 2001.

[24] D. Jackson and C.A. Damon, "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector," *IEEE Trans. Software Eng.*, vol. 22, no. 7, pp. 484-495, July 1996.

[25] D. Jackson, I. Scheckter, and I. Shlyakhter, "Alcoa: the Alloy Constraint Analyzer," *Proc. Int'l Conf. Software Eng.*, pp. 730-733, June 2000.

[26] S. Khurshid and D. Marinov, "Checking Java Implementation of a Naming Architecture Using TestEra," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 3, 2001.

[27] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson, "A Case for Efficient Solution Enumeration," *LNCS*, vol. 2919, pp. 272-286, 2004.

[28] S. Khurshid and C. Pasareanu, "Generalized Symbolic Execution for Model Checking and Testing," *Proc. Int'l Conf. Tools and Algorithms for Construction and Analysis of Systems*, Apr. 2003.

[29] J.C. King, "Symbolic Execution and Program Testing," *Comm. ACM*, vol. 19, no. 7, pp. 385-394, July 1976.

[30] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Trans. Software Eng.*, vol. 12, pp. 96-109, Jan. 1986.

[31] I. MacColl, D. Carrington, and P. Stocks, "An Experiment in Specification-Based Testing," Technical Report 96-05, Software Verification Research Centre, Dept. of Computer Science, The Univ. of Queensland, May 1996.

[32] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, "An Evaluation of Exhaustive Testing for Data Structures," Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, Mass., Sept. 2003.

[33] D. Marinov and S. Khurshid, "TestEra: A Novel Framework for Automated Testing of Java Programs," *Proc. Int'l Conf. Automated Software Eng.*, Nov. 2001.

[34] L. Morell, "A Theory of Fault-Based Testing," *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 844-857, Aug. 1990.

[35] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proc. Design Automation Conf.*, June 2001.

[36] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," *Proc. Int'l Conf. Unified Modeling Language*, Oct. 1999.

[37] S. Rapps and E. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 367-375, Apr. 1985.

[38] M. Saaltink, "The Z/EVES System," *LNCS*, vol. 1212, pp. 72-85, 1997.

[39] E. Schwartz, "Design and Implementation of Intentional Names," master's thesis, MIT Laboratory for Computer Science, Massachusetts Inst. of Technology, Cambridge, June 1999.

[40] L. Shlyakhter, "Generating Effective Symmetry-Breaking Predicates for Search Problems," *Proc. Workshop Theory and Applications of Satisfiability Testing*, June 2001.

[41] J.M. Spivey, *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.

[42] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Trans. Software Eng.*, vol. 22, no. 11, pp. 777-793, Nov. 1996.

[43] K.J. Sullivan, J.B. Dugan, and D. Coppit, "The Galileo Fault Tree Analysis Tool," *Proc. Int'l Fault-Tolerant Computing Symp.*, pp. 232-235, June 1999.

[44] K.J. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software Assurance by Bounded Exhaustive Testing," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 133-142, July 2004.

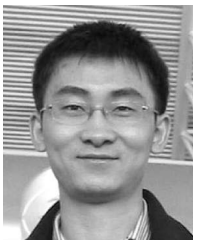
[45] Sun Microsystems, Java 2 platform, standard edition, v1.3.1 API Specification, http://java.sun.com/j2se/1.3/docs/api/, 2001.

[46] W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl, "Fault Tree Handbook," Technical Report NUREG-0492, US Nuclear Regulatory Commission, Washington, D.C., 1981.

[47] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Trans. Software Eng.*, vol. 6, no. 3, pp. 236-246, May 1980.

[48] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computer Surveys*, vol. 29, no. 4, pp. 366-427, Dec. 1997.

David Coppit received the Master's and PhD degrees in computer science from the University of Virginia in 1998 and 2003, respectively. He received his undergraduate degrees in computer science and physics from the University of Mississippi in 1995. He is an assistant professor of computer science at the College of William and Mary. His research interests include software development methods, software plans, applied formal methods, and software verification. He is a member of the IEEE.



Jinlin Yang received the MCS degree in computer science from the University of Virginia in 2004 and the BE degree in computer science and technology from the Tsinghua University, P.R. China in 2001. He is a PhD candidate in the Department of Computer Science at the University of Virginia. His research interests include program analysis, program testing and verification, software evolution, and model checking. He is a member of ACM and ACM SIGSOFT.

Sarfraz Khurshid received the PhD degree in computer science at the Massachusetts Institute of Technology in 2004. He received the BSc degree in mathematics and computer science from Imperial College London in 1994, and completed Part III of the Mathematical Tripos at Trinity College Cambridge in 1998. He is an assistant professor in the Electrical and Computer Engineering Department at the University of Texas at Austin, where he leads the Software Testing and Verification Group at the Center for Advanced Research in Software Engineering. His current research focuses on software testing, specification languages, code conformance, model checking, and applications of heuristics in program analysis.

Wei Le received the master's degree at the University of Virginia in 2005 and the bachelor's degree at Zhejiang University, China in 2002, both in computer science. She is a graduate student in computer science at the University of Virginia. Her research focuses on software testing, software security, and program analysis. She is a member of the ACM.

Kevin Sullivan received the PhD degree in computer science and engineering in 1994 from the University of Washington. He is an associate professor and VEF Faculty Fellow in Computer Science at the University of Virginia. His interests are in the structures, properties, and value of design, with a particular emphasis on issues of modularity and integration in software design, and dependability as a user-perceived property. He is a senior member of the IEEE and a member of the ACM and ACM SIGSOFT.