# Generating Test Inputs for Fault-Tree Analyzers using Imperative Predicates

Saša Misailović Aleksandar Milićević
University of Belgrade
Belgrade, Serbia
{sasa.misailovic,aca.milicevic}@gmail.com

Sarfraz Khurshid
University of Texas
Austin, TX 78712
khurshid@ece.utexas.edu

Darko Marinov
University of Illinois
Urbana, IL 61801
marinov@cs.uiuc.edu

## Abstract

*This paper presents a case study on how Korat can be used in system testing, specifically in testing a large fault-tree analyzer developed for NASA. A fault-tree analyzer takes as input a fault tree that models how combinations of failures in the components of a system produce overall failures of the system. Testing a fault-tree analyzer requires generating fault trees. Korat is a previously developed testing tool that automates generation of structurally complex test inputs. Fault trees are structural in that they can be represented as graphs, and the nodes in the graphs need to satisfy certain complex constraints. Korat allows the user to express these constraints in widely used imperative programming languages such as Java. Previous research has shown how to test a fault-tree analyzer using another testing tool that requires the user to express constraints in a declarative language. This paper compares these two approaches. The results show that Korat generates a larger number of inputs but does not prune out non-equivalent inputs and thus can generate inputs that reveal errors in the system under test.*

## 1 Introduction

A *fault-tree* [41] models a system failure. Given a fault-tree, a *fault-tree analyzer* computes the likelihood that the system will fail over a given period of time. Fault-trees and analyzers are typically used in mission critical systems. Therefore, incorrect behavior of an analyzer can be very costly. Checking the correctness of an analyzer itself is hard. An analyzer typically has tens of thousands of lines of code that performs complex computations on fault trees. Checking rich correctness properties for such a large piece of code is beyond the reach of most current state-of-the-art tools for automated testing.

In previous work [18,40], we showed how constraint-based testing can be used to systematically test fault-tree analyzers. We used the TestEra tool [25] to test two fault-tree analyzers, Galileo [9] and NOVA [8]. TestEra is a tool that automates generation of structurally complex test inputs and is thus suitable for generation of fault trees. Fault trees are structural in that they can be represented with graphs, and the nodes in the graphs need to satisfy certain complex constraints.

TestEra requires the user to describe the constraints of desired test inputs as a formula in a relational, first-order logic with transitive closure [16]. We call such formulas *declarative predicates* since they are written in a declarative language. TestEra also requires the user to provide a bound on the input size. TestEra then automatically (1) leverages the SAT-based Alloy tool-set [16] to enumerate solutions for the formula within the given bound and (2) translates each solution into an input for the program under test. TestEra performs *bounded-exhaustive testing* by testing a program on all inputs up to the given size. TestEra's analysis of Galileo and NOVA revealed several subtle errors in both analyzers [40].

While TestEra successfully revealed errors in actual implementations of the two fault-tree analyzers, TestEra required the user to write constraints in a declarative language (first-order logic with extensions), which has a different syntax and semantics from commonly used programming languages and thus can pose a learning challenge. Writing constraints for enumerating fault trees as test inputs poses an additional challenge. Namely, there exist a very large number of fault-trees even for very small sizes. This necessitates that the user must write additional constraints that focus generation to a smaller set of test inputs to make testing feasible. *Symmetry-breaking constraints* are a common form of additional constraints, which rule out generation of isomorphic inputs. Manually writing symme-

try breaking constraints for complex inputs is tedious and itself error-prone. Automatic synthesis of symmetry breaking constraints is possible in principle but known SAT-based approaches [33, 34] do not scale for constraints that represent complex inputs—the resulting SAT formulas tend to be impractically large [19].

The contribution of this paper is to use a solver for *imperative predicates* to generate test inputs for fault-tree analyzers. We use Korat, a solver that we have previously developed and used mostly for testing smaller program units [2, 24, 28]. The Korat tool is available for public download at `http://mir.cs.uiuc.edu/korat`. Unlike TestEra that requires the user to write the constraints on test inputs as the first-order logic formulas, Korat allows the user to write the constraints in a widely used programming language such as Java. Since Java is an imperative language, we call the constraints written in it *imperative predicates*. Like TestEra, Korat requires the user to provide a finitization that bounds the desired test input size. Also like TestEra, Korat automatically generates all test inputs (within the bounds) that satisfy the constraints. Korat performs a systematic search of the predicate's input space and attempts to prune equivalent inputs. Previous research has shown how to use Korat for unit testing and how to optimally prune some test inputs.

A key issue in generation of fault trees is that they are not actually trees (in the sense of graph theory) but directed acyclic graphs (DAGs): if a subsystem is used in two different ways in the system modeled by a fault tree, then the subsystem is shared in the "tree". When test inputs are indeed trees, Korat can prune away all *non-isomorphic* test inputs, i.e., test inputs that differ only in the identity of the nodes in the trees but have the same shape of the trees and the same elements [2]. Korat is reimplemented in the Spec Explorer tool [3] at Microsoft and has been successfully used for detecting errors in several applications (for example in a production-quality XPath compiler [38]). Scalability of bounded-exhaustive testing tools does not depend as much on the complexity/size of the tested code as it depends on the complexity of inputs that the code operates on. In all previous applications, the underlying structure of test inputs was actual trees (for example, parse trees of XML documents). This paper considers how Korat works for DAGs. Our results show that Korat generates a larger number of inputs than TestEra but does not prune out non-equivalent inputs.

## 2 Fault Trees

Fault trees [41] model system failures; a fault tree represents the overall failure of a system as a combination of failures of basic components of the system. A *static* fault tree models how boolean combinations of component-level failure events produce system failures. Every fault tree has a *top-level* event that represents system level failure (and is graphically drawn as the root of the tree). Figure 1(a) illustrates a static fault tree. The interior nodes of a tree are boolean *gates* and the leaves are *basic events*. The failure of basic events is characterized probabilistically by *basic event model* that consists of:

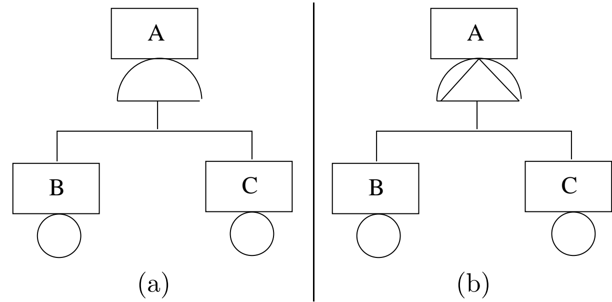


**Figure 1. (a) Static fault tree; system level failure is an AND gate: if both events B and C occur, the system fails. (b) Dynamic fault tree; system level failure is a priority AND gate: if first event B occurs and then event C occurs, the system fails.**

- the *rate parameter* (*lambda*) that defines the exponential distribution that characterizes the basic event's failure;
- the *coverage model* that defines the probabilities that the component masks an internal failure (*res*), that a component fails in a way that can be detected by the system (*cov*), and that the component fails and brings down the system (*sing*); the sum of these three probabilities is 1; and
- the *dormancy value* (*dorm*) that defines the probability of failure if the basic event is used as an input to a gate to replace one of its failed inputs.

A basic event may have a *replication value* (*repl*), which allows the event to represent identical events connected to the same location(s) in a fault tree.

*Dynamic* fault trees augment static fault trees with constructs that allow modeling fault-tolerant systems; these constructs allow modeling, for example, how a sequence of events causes failure, and functional dependencies, such as failure of a trigger event causes failure of all dependent events. Figure 1(b) illustrates a dynamic fault tree.

Fault trees can be represented graphically or with a *fault tree grammar* [8]. For example, the tree in Figure 1(b) together with its basic event model can be textually represented as:

```
toplevel A;
A pand B C;
B lambda=.01 cov=.75 res=0 repl=1 dorm=.25;
C lambda=.05 cov=.5 res=0.1 repl=1 dorm=.5;
```

A *fault tree analyzer* computes, for a given fault tree and its failure models, the probability of system-level failures.

### 2.1 Galileo and NOVA

In previous work [18, 40], bounded-exhaustive testing was applied on two fault-tree analyzers, namely Galileo [9] and NOVA [8]. They take as inputs strings in fault-tree grammar. Galileo is a highly optimized, dynamic fault-tree analyzer developed for NASA and is in production use. The focus was on testing the core analysis functionality of Galileo (excluding the user interface or advanced modeling features); the core itself consists of 10,680 non-comment non-blank lines of code.

NOVA is a more recently developed fault-tree analyzer and has been based on a formal Z specification [37] with the aim of using NOVA as a test oracle for checking other fault-tree analyzers. Thus, NOVA was made as simple as possible to reduce the potential that NOVA itself has a fault. NOVA enables *differential testing* of fault tree analyzers: run NOVA and another analyzer on the same set of test inputs, and check if there is any test input for which the outputs differ.

## 3 Generating fault-trees as test inputs

Systematic testing of a fault-tree analyzer requires enumeration of (a large number of) fault-trees as test inputs. Since fault-trees have complex structural constraints, a string that represents a valid fault-tree cannot feasibly be generated at random, so a systematic approach for generation is needed.

In previous work [18, 40], we explored such an approach using the test generation tool TestEra: a first-order logic formula captured the desired structural constraints, which were solved using an off-the-shelf SAT solver, and each solution was translated into a string in fault-tree grammar. The resulting test suite uncovered several errors in both Galileo and NOVA. A key factor that enabled TestEra to feasibly test these analyzers was generation of non-isomorphic inputs. However, TestEra requires users to manually write constraints, so called *symmetry-breaking predicates* that rule out generation of isomorphic inputs; such constraints are usually quite complex and writing them is error-prone.

Generation of fault trees in previous work [18, 40] used several steps to produce actual strings in the fault-tree grammar from structural constraints on the desired fault trees. In this paper, we focus on the crucial first step that consists of generation of *abstract fault trees* (AFTs) [40]. Each AFT represents a skeleton of a fault tree: it represents the structure of the tree that consists of the interior nodes, called *gates*, and the leaves, called *basic events* (Section 2). AFTs do not contain information about *basic event model*. Instead, the second step post-processes AFTs to add that information and to create concrete fault trees. Finally, the concrete fault trees are printed from the internal format into the actual string inputs for the tools. More details about these steps are available in [40].

### 3.1 Abstract fault trees

Figure 2 shows code that can represent abstract fault trees and check their structural constraints. Each object of class `AFT` is an abstract fault tree, which has `toplevel` events as its root, and the total number of events is `size`. Each object of class `Event` represent an event. For brevity of code, we use the same class to encode both basic events and gates. Gates are the internal nodes and can have a number of children nodes stored in the array `tail`.

In general, one can create an arbitrary object graph consisting of `AFT` and `Event` nodes (subject to typing constraints). The methods `repOK` check that such a graph indeed encodes a *valid* abstract fault tree. (We follow Liskov [23] in using the name `repOK` for the methods that check representation invariants of data structures that implement abstract data types.) These `repOK` methods are *imperative predicates* that check the desired structural constraints of the AFTs. As mentioned earlier, (abstract) fault trees are not necessarily trees but allow structures that have sharing between nodes. Effectively, the methods `repOK` check that the object graph is a connected, rooted, directed, and acyclic graph. The methods use a standard depth-first traversal of the graph to check this property. The methods keep the set of already traversed events in `visited` and keep the set of nodes traversed along one path in `stack` to ensure that there are no direct cycles. This code is simple enough that an undergraduate student researcher (the second paper author) produced it in a few hours.

Korat [2, 24, 28] is our tool that can generate object graphs that satisfy the properties expressed with

```
public class AFT {
  private Event toplevel;
  private int size;

  public boolean repOK() {
    if (toplevel == null)
      return (size == 0);
    Set<Event> visited = new HashSet<Event>();
    visited.add(toplevel);
    if (!toplevel.repOK(visited, new Stack<Event>()))
      return false;
    return visited.size() == size;
  }
}

public class Event {
  private Event[] tail; // if length is 0, it's a basic event;
                        // otherwise, it's a gate
  public boolean isBasic() { return tail.length == 0; }

  public boolen repOK(Set<Event> visited,
        Stack<Event> stack) {
    stack.push(this);
    for (int i = 0; i < tail.length; i++) {
      Event e = tail[i];
      for (int j = 0; j < i; j++)
        if (e == tail[j])
          return false;
      if (e.isBasic()) {
        visited.add(e);
      } else {
        if (stack.search(e) != -1) // e is on stack
          return false; // directed cycle
        if (visited.add(e)) {
          if (!e.repOK(visited, stack))
            return false;
        }
      }
    }
    stack.pop();
    return true;
  }
}
```

**Figure 2. Code that represents abstract fault trees and checks their structural properties.**

imperative predicates given in the `repOK` methods. Besides `repOK` methods, Korat takes as input a *finitization* that bounds the size of the graphs. Figure 3 shows the finitization for AFTs. It bounds the total number of events in each tree and the size of the tree. The finitization uses the Korat library classes and methods that construct sets of objects and assign these objects to fields.

Given specific values for the bounds, Korat can generate all valid AFTs within the given bound. As mentioned, translating all these structures into concrete test inputs and using them to test code is called *bounded-exhaustive testing*. Clearly, it is not necessary to exhaustively use the tests, and one can always choose to sample in some way, either randomly or guided by some test coverage criteria. However, using all possible inputs with the given bounds provides a guarantee that there is no error in the code under test, within the bounds.

```
public static IFinitization finAFT(int numEvents,
        int minSize, int maxSize) {
    IFinitization f = FinitizationFactory.create(AFT.class);
    IObjSet events = f.createObjSet(Event.class, numEvents);
    IArraySet tails = f.createArraySet(Event[].class,
           f.createIntSet(0, numEvents),
           events, numEvents);
    f.set("toplevel", events);
    f.set("size", f.createIntSet(minSize, maxSize));
    f.set("Event.tail", tails);
    return f;
}
```

**Figure 3. Finitization that bounds the size of the generated abstract fault trees, giving the total number of events in the tree and the bounds for the tree size.**

Korat can write the generated objects into files or graphically show them. For example, the command:

```
java korat.Korat --visualize --class AFT --args 3,3,3
```

instructs Korat (1) to generate all AFTs with exactly three nodes and (2) to display graphically the generated structures. Figure 4 shows the visualization of some examples of generated structures. The visualization in Korat was inspired by Alloy [16], the language used for TestEra. Our current Korat implementation [28] leverages the Alloy Analyzer's visualization facility as Korat automatically translates object graphs into the Alloy representation. The visualization can assist users in writing correct `repOK` methods and also in understanding any errors revealed in the code tested with the Korat-generated inputs.

### 3.2 Non-equivalent test inputs

It is desirable to test the code only with non-equivalent test inputs: if two test inputs either both expose the same error or none exposes the error, we need to test the code with only one of the two inputs. It is hard in general to know a priori which test inputs expose (the same) errors: omitting any test input from a test suite may omit the only input that exposes an error. However, *isomorphic* inputs are almost always equivalent [24]. We call two inputs, represented as Java object graphs, isomorphic if and only if they have the same shape of the structure and the same primitive values but differ only in the identity of the objects in the structure. Consider, for example, the first AFT in Figure 4. Reordering the identity of the nodes $\texttt{Event}_1$, $\texttt{Event}_2$, and $\texttt{Event}_3$ (while keeping the same shape of the graph), results in isomorphic AFTs.

Korat automatically generates only non-isomorphic structures when test inputs are rooted and deterministically edge-labeled (object) graphs. (Details of the

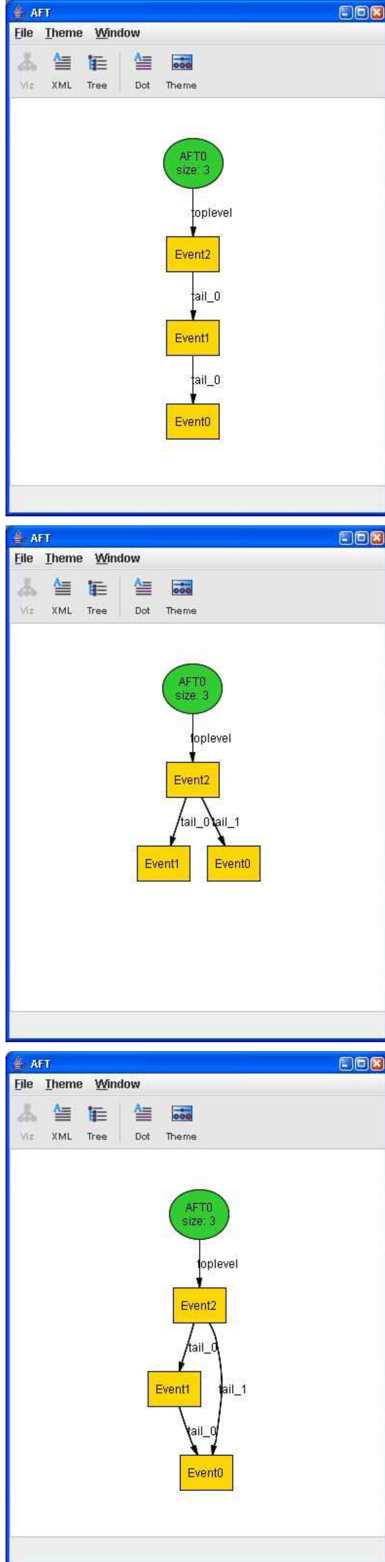


**Figure 4. Example Korat-generated abstract fault trees with three events.**

Korat algorithm and the proof of non-isomorphic generation are presented in [24]). However, Korat can generate some isomorphic structures when the inputs are general graphs, i.e., a node/object can have several outgoing edges/fields with the same label. At the concrete level, object graphs in Java do not have such nodes since objects cannot have repeated field names. However, there can be such nodes if inputs are not rooted (in which case, we can introduce a special root node and connect it to every other node with edges that share the same label) or more broadly, when we view sets at the abstract level (such that one objects can have fields that are sets of objects). The latter is the case for AFTs: every `Event` has effectively a *set* of children nodes, but in our example code this set is at the concrete level stored in the array `tail`. Therefore, Korat treats different orders of elements in the array as different inputs, even when they represent the same set.

In theory, TestEra can automatically generate only non-isomorphic structures even for general graphs. For general graphs with $n$ nodes, thus, TestEra could automatically generate up to $n!$ less graphs than Korat, and hence TestEra would be more practical than Korat when inputs are general graphs. To generate non-isomorphic inputs, TestEra can use automatic symmetry breaking [33], but breaking all symmetries can significantly slow down TestEra's generation (and would correspond to generating all inputs with Korat and then filtering out isomorphic ones). In practice, TestEra automatically breaks some symmetries for structures and general graphs.

Experimental results for several structures [18] show that the number of structures that TestEra with default symmetry breaking generates is from 1.5 times to 177 times larger than the actual number of non-isomorphic structures. The user of TestEra can manually provide symmetry-breaking predicates to efficiently break all symmetries, for example following the methodology that we previously proposed [19] which works when structures have a tree backbone [29]. However, it is not clear how to break all symmetries for more general graphs, such as AFTs, when structures have no tree backbone.

The previous study [40] on fault trees used TestEra to generate test inputs for bounded-exhaustive testing. A TestEra expert (the third paper author) developed the constraints for AFTs and also a symmetry-breaking predicate that eliminates most isomorphic AFTs. In this study, we develop a corresponding Java predicate for this constraint, and we use Korat for generation. Our use of Korat has revealed that the symmetry-breaking predicate written for TestEra was incorrect:

| size | time [s] | explored | generated | non-isom. |
|---|---|---|---|---|
| 1 | 2.0 | 1 | 1 | 1 |
| 2 | 2.1 | 5 | 1 | 1 |
| 3 | 2.2 | 24 | 4 | 3 |
| 4 | 2.3 | 288 | 57 | 16 |
| 5 | 3.0 | 17767 | 3399 | 164 |
| 6 | 239.9 | 6177272 | 1026729 | 3341 |

**Figure 5. Korat's generation of abstract fault trees of various sizes.**

it eliminated not only non-isomorphic inputs but also some valid inputs! Thus, TestEra could miss some errors in the fault-tree analyzers Galileo and NOVA. This anecdotal evidence shows that manually breaking symmetries is an error-prone task even for Alloy/TestEra experts.

### 3.3 Results

Figure 5 shows the results of Korat's generation of abstract fault trees. For AFTs of various size, we tabulate the time that Korat took for generation, the number of structures that Korat internally explored during search (some of which are not valid AFTs), the number of AFTs that Korat actually generated as its output (all of which are valid AFTs), and the number of non-isomorphic AFTs among those generated by Korat. We have performed all timing experiments on a Pentium III 866MHz machine running Sun's JVM 1.5.0_07. (We limited each run to 800MB of memory, but note that the amount of memory that Korat uses does not grow with the length of the search since Korat does not perform a stateful search.)

Korat's search generates *candidate structures* and invokes `repOK` on them to check their validity [24]. If `repOK` returns `true`, Korat generates the structure. Note that when `repOK` returns `true`, it means that the structure is valid and can be used as a test input for the code under test; it does not mean that this particular test input will reveal an error. Korat then creates the next candidate structure based on the monitoring of dynamic execution of `repOK`. The column with the number of explored structures presents how many candidates Korat tried while generating the valid AFTs. Note that the ratio of explored and generated structures is fairly small, around 6 times, attesting to the quality of Korat's search.

The last column shows the number of non-isomorphic AFTs among those that Korat generates. We use the nauty tool [27] to count the number of non-isomorphic AFTs. The sequence of non-isomorphic AFTs appears in the Sloane's Encyclopedia of Integer Sequences [36] (as a subsequence in A122078), which gives some confidence that the `repOK` methods are correct. (We have customarily used the Encyclopedia to verify the number of structures that Korat generates [2]). Our inspection of all graphs for several smaller sizes shows that Korat indeed does not miss any (non-isomorphic) AFT, which further increases our confidence. However, Korat does generate a large number of isomoprphic AFTs. For AFTs with 6 nodes, for example, Korat generates over a million of AFTs, but only 3,341 of them are non-isomorphic.

For comparison, TestEra generates a smaller number of AFTs for all sizes [40, page 6, Table 1 for rows with Seqs and FDeps being 0]: 3 for size 3; 16 for size 4; 176 for size 5; 4,229 for size 6; and 230,470 for size 7. (We estimate that Korat would generate over $10^{10}$ AFTs for size 7, which would require quite a lot of time.) This means that TestEra (*with* manually written symmetry breaking) can generate larger AFTs that Korat (*without* manually written symmetry breaking) can generate. However, as already mentioned, we have found that TestEra misses some valid AFTs. The reason is that the manually written symmetry-breaking predicates for TestEra eliminate some non-isomorphic AFTs.

In summary, our results show that Korat can generate structures that are not trees and can generate test inputs that can detect errors in important, real applications. The results also show some complementary features of Korat and TestEra. In parciular, it is worthwhile to investigate how to combine the *automatic, full* isomorphism elimination that Korat does for *some* structures with the *manual, partial* isomorphism elimination that TestEra does for *all* structures. The methodologies from these tools could enhance each other, enabling automatic breaking of (1) some symmetries for general graphs in Korat and (2) all symmetries for Java-like object graphs in TestEra (without slowing down the generation by generating too many symmetry-breaking predicates). We leave this investigation as future work.

## 4 Related work

Using constraints to represent inputs is not a new idea and dates back at least three decades [7,15,20]; the idea has been implemented in various tools including EFFIGY [20], TEGTGEN [21], and INKA [11]. But the focus of prior work has been to solve constraints on primitive data, such as integers and booleans, and not to solve constraints on complex structures, which requires very different constraint solving techniques.

Korat [2, 24] and TestEra [18, 25] are among the first frameworks to support constraint-based generation of complex structures.

One of the earliest works to emphasize the importance of specification-based testing is by Goodenough and Gerhart [10]. Various projects automate test case generation from specifications, such as Z specifications [14,37,39], UML statecharts [30,31], ADL specifications [4,32], or AsmL specifications [12]. These specifications typically do not involve structurally complex inputs.

The first version of the Spec Explorer tool (then called AsmLT) [12] was internally based on finite-state machines (FSMs): an AsmL [13] specification is transformed into an FSM, and different traversals of FSM are used to construct test inputs. Korat adds structure generation to the generation based on finite-state machines [12]. AsmLT was successfully used for detecting faults in a production-quality XPath compiler [38].

There are many tools that produce test inputs from a description of tests. QuickCheck [6] is a tool for testing Haskell programs [17]. It requires the tester to write Haskell functions that can produce valid test inputs; executions of such functions with different random seeds produce different test inputs. DGL [26] and lava [35] generate test inputs from production grammars. They were used mostly for random testing, although they can also systematically generate test inputs. However, they cannot easily represent structures with complex invariants. Even though DGL is Turing-complete and in theory it is possible to specify complex structures, doing so for a structure would essentially be the same as (and require as much effort as) writing a dedicated generator for that particular structure.

Cheon and Leavens developed jmlunit [5] for testing Java programs. They use the Java Modeling Language (JML) [22] for specifications; jmlunit translates JML specifications into test oracles for JUnit [1]. This approach automates execution and checking of methods. However, the burden of test generation is still on the tester who has to provide sets of possibilities for all method parameters and construct complex data structures using a sequence of method calls.

## 5 Conclusions

This paper has presented how Korat can be used to test a large piece of code such as the the core analysis functionality of the Galileo fault-tree analyzer that has over 10,000 lines of C++ code. Korat is a tool that can generate structurally complex test inputs and is suitable for generating fault trees that are the test inputs for fault-tree analyzers. Korat requires the user to provide an imperative predicate (written in a common programming language such as Java) that specifies the desired structural constraints of test inputs and a finitization that bounds the desired size of test inputs. Korat generates all test inputs (within the bounds) that satisfy the constraints. Using such test inputs constitutes bounded-exhaustive testing. Previous research has shown how to use the TestEra tool, based on declarative predicates, for bounded-exhaustive testing of Galileo. Our results show that Korat and TestEra are complementary in their current uses: Korat generates a larger number of inputs than TestEra, but Korat does not prune out non-equivalent inputs. These initial results are positive, and the future work is to explore how to prune away non-equivalent inputs in Korat.

## Acknowledgments

We thank Dragan Milićev for discussions on the initial design of the current Korat implementation, Nemanja Petrović for discussions on generating graphs with Korat, and Vladeta Jovović for sharing with us his expertise in using the Sloane's Encyclopedia. This material is based upon work partially supported by the NSF under Grant Nos. 0438967, 0613665, and 0615372. We also acknowledge support from Microsoft Research.

## References

[1] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[3] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-based testing of object-oriented reactive systems with Spec Explorer. Technical report, Microsoft Research, May 2005. `http://research.microsoft.com/specexplorer/`.

[4] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proc. 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Sept. 1999.

[5] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, June 2002.

[6] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *Proc. ACM SIGPLAN workshop on Haskell*, 2002.

[7] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, Sept. 1976.

[8] D. Coppit. *Engineering Modeling and Analysis: Sound Method and Effective Tools*. PhD thesis, The University of Viginia, Charlottesville, VA, 2003.

[9] D. Coppit and K. J. Sullivan. Galileo: A tool built from mass-market applications. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[10] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.

[11] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Clearwater Beach, FL, 1998.

[12] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[13] Y. Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*. Oxford University Press, 1995.

[14] H.-M. Horcher. Improving software tests using Z specifications. In *Proc. 9th International Conference of Z Users, The Z Formal Specification Notation*, 1995.

[15] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3), 1975.

[16] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

[17] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003.

[18] S. Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Dec. 2003.

[19] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A case for efficient solution enumeration. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003.

[20] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[21] B. Korel. Automated test data generation for programs with procedures. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, San Diego, CA, 1996.

[22] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.

[23] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[24] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[25] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.

[26] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4), July 1990.

[27] B. D. McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 1, pages 45–87. 1981.

[28] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proc. ICSE Research Demos (ICSE Demo 2007)*, Minneapolis, MN, May 2007. (To appear.).

[29] A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. SIGPLAN Conference on Programming Languages Design and Implementation*, Snowbird, UT, June 2001.

[30] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. Second International Conference on the Unified Modeling Language*, Oct. 1999.

[31] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, 1998.

[32] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., Mountain View, CA, Apr. 1994.

[33] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.

[34] I. Shlyakhter. *Declarative Symbolic Pure Logic Model Checking*. PhD thesis, MIT, February 2005.

[35] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In *Proc. 2nd conference on Domain-specific languages*, 1999.

[36] N. J. A. Sloane, S. Plouffe, J. M. Borwein, and R. M. Corless. The encyclopedia of integer sequences. *SIAM Review*, 38(2), 1996. `http://www.research.att.com/~njas/sequences/Seis.html`.

[37] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[38] K. Stobie. Advanced modeling, model based test generation, and Abstract state machine Language (AsmL). Seattle Area Software Quality Assurance Group, `http://www.sasqag.org/pastmeetings/asml.ppt`, Jan. 2003.

[39] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11), 1996.

[40] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[41] United States Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981. NUREG-0492.