WISE: Automated Test Generation for Worst-Case Complexity

Jacob Burnim   Sudeep Juvekar   Koushik Sen

EECS Department, UC Berkeley, CA, USA.

{jburnim,sjuvekar,ksen}@cs.berkeley.edu

Abstract

Program analysis and automated test generation have primarily been used to find correctness bugs. We present complexity testing, a novel automated test generation technique to find performance bugs. Our complexity testing algorithm, which we call WISE *(Worst-case Inputs from Symbolic Execution), operates on a program accepting inputs of arbitrary size. For each input size,* WISE *attempts to construct an input which exhibits the worst-case computational complexity of the program.* WISE *uses exhaustive test generation for small input sizes and generalizes the result of executing the program on those inputs into an "input generator." The generator is subsequently used to efficiently generate worst-case inputs for larger input sizes. We have performed experiments to demonstrate the utility of our approach on a set of standard data structures and algorithms. Our results show that* WISE *can effectively generate worst-case inputs for several of these benchmarks.*

1. Introduction

Automated test generation has been an area of active research for more than three decades [20, 6, 23]. Many techniques have been developed to automatically generate test inputs, both by treating programs as black boxes [3, 24, 10] and by examining the structure of programs [20, 7, 21, 15, 19, 28, 4, 29, 2, 9, 11, 25, 5]. The goal of most of these automated testing techniques has been to find bugs and to improve confidence in the correctness of software. Little work has focused on generating test inputs that reveal performance bottlenecks.

We present a novel automated test generation technique to find performance problems in a program unit. Specifically, given a program that accepts inputs of arbitrary size, our technique attempts to construct a test input of each possible size which exhibits the program's worst-case computational complexity. For example, given a quicksort algorithm which runs on integer arrays, our technique will construct arrays of length $n = 1, 2, \ldots$, for which the quicksort computation requires a number of steps proportional to n^2,

highlighting the worst-case $O(n^2)$ complexity of quicksort. Since our technique tests the computational complexity of a program unit, we call it a technique for *computational complexity testing*, or simply *complexity testing*.

Complexity testing has several uses:

- We can check if an implementation of an algorithm matches the theoretical worst-case computational complexity. If complexity testing shows that the implementation has complexity worse than the theoretical bound, then the implementation has a performance bug and does not conform to its algorithmic specification.

- If we are designing a new algorithm, we can perform a quick complexity testing of the algorithm and discover its worst-case computational complexity without going into a manual computational complexity analysis. Thus, complexity testing could help programmers to discover inherent performance bottlenecks in their algorithms without requiring any sophisticated knowledge of the underlying theory.

- Since our technique yields a concrete test input showing a worst-case execution of a program, it can aid in debugging performance problems and understanding the cause of worst-case executions.

Our complexity testing technique, Worst-case Inputs from Symbolic Execution (WISE), is based on symbolic test generation. A naïve algorithm for using symbolic test generation to discover worst-case input of size $n > 0$ would be to symbolically enumerate all feasible execution paths of the program for inputs of size n and then create an input for the longest feasible execution path. Although this naïve algorithm works for inputs of small sizes, it fails to scale for inputs of larger sizes because the number of feasible execution paths typically increases at least exponentially with an increase in the size of inputs. For example, our symbolic test generation tool could explore in a few minutes all feasible execution paths of an implementation of the Bellman-Ford algorithm for inputs of size 4, but for inputs of size 5 the tool failed to terminate even after 24 hours.

Our complexity testing technique WISE is an attempt to significantly reduce this scalability problem. *The technique*

is based on the insight that if we run our naïve algorithm on small input sizes, we should be able to learn some concept or rule about the program that will help us to prune the search for worst-case inputs of larger sizes. For example, in case of an unbalanced binary search tree, we can learn the concept that if we iteratively add elements larger than previously added elements, then we get a list instead of a tree. The complexity of search in such a tree would be linear instead of $O(\log n)$. We call these concepts *generators*. Once we have computed a generator, we can use it to efficiently generate worst-case inputs for larger input sizes.

In this paper, we propose a simple class of generators called *branch policy generators*. These generators restrict the conditional branches that may be included in an execution path, e.g. by allowing only the *"true"* branch to be taken for a particular conditional statement. We give a procedure for computing a branch policy given the results of exhaustive test generation on small input sizes. Further, we show that, under certain conditions, this procedure is *sound* for complexity analysis—i.e. that the computed generator does not prune away all worst-case executions for any input size. More precisely, we prove that, for each program P, for sufficiently large N the exhaustive search for all inputs of size up to N will yield a *sound* generator.

We evaluate our technique on several Java benchmarks, including a Quicksort routine, a red-black tree search, and the Bellman-Ford graph algorithm. In our experiments, the sufficiently-large input size N is less than 10 in all cases, and the produced branch policy generators are effective in pruning the space of program executions for large input sizes. Our experiments demonstrate that WISE can effectively find large worst-case inputs for real algorithms and data structure operations.

2. Overview

In this section, we give an informal overview of the WISE algorithm using the example in Figure 1. The example is a Java implementation of insertion into a sorted linked list. The code includes a driver which creates a new, empty list and inserts N integers into it.

The WISE algorithm works in the following three stages:

1. First, WISE uses symbolic execution to generate test inputs for all feasible execution paths of the program when run on $N = 1, 2,$ or 3.

2. Second, from these execution paths, WISE extracts a *generator G* (which we will define later), an entity that characterizes a small subset of execution paths including a worst-case execution path of the program.

3. Finally, WISE uses *guided* symbolic test generation to produce test inputs of larger sizes only for the paths

```
1   class List {
2       int x;   List next;
3       static final int SENTINEL = Int.MAX;
4       private List(int x, List next) {
5           this.x = x;
6           this.next = next;
7       }
8       public List() {
9           this(SENTINEL, null);
10      }
11      public void Insert(int data) {
12          if (data > this.x) {
13              next.Insert(data);
14          } else {
15              next = new List(x, next);
16              x = data;
17          }
18      }
19      public static void main(int N) {
20          List list = new List();
21          for (int i = 0; i < N; i++)
22              list.Insert(Input());
23      }
24  }
```

Figure 1. Sorted linked list.

conforming to the generator G. For each larger input size, WISE outputs a worst-case generated test input.

We next describe these stages on the example in Figure 1.

In the first stage of the algorithm, we use symbolic execution and constraint solving to enumerate all feasible execution paths of the program. These feasible execution paths for $N = 3$ are shown in the form of a computation tree in Figure 2, where $x_1, x_2,$ and x_3 are the three symbolic inputs to the program. Each node in the tree corresponds to a statement in the Insert function in Figure 1, where the nodes for conditional statements are labeled with predicates on the inputs which determine whether the "true" or "false" branch of the conditional is taken. For simplicity, we show only the conditional statements at line 12 in Figure 2. The label on the edge originating from a node represents the value of the conditional statement during the execution of the program. The dotted edges represent infeasible branches. There are exactly $3! = 6$ feasible execution paths in the tree, each corresponding to a unique ordering of x_1, x_2 and x_3. The longest feasible execution path (i.e. the path containing the largest number of conditional statements) is shown in bold.

In the second stage of the algorithm, after finding a longest execution paths for each $N = 1, 2, 3$, we observe that these longest paths obey the following simple rule:

- The "true" branch of the conditional at line 12 *must* always be taken if the branch is *feasible*. Equivalently, the "false" branch of this conditional may be taken only when the "true" branch is *infeasible*.
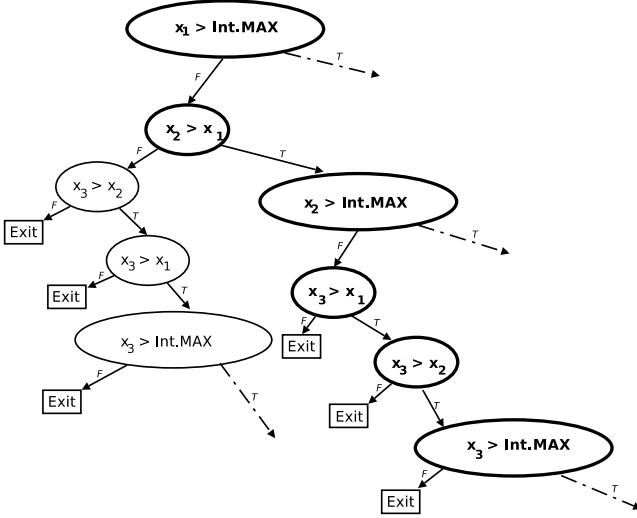
Figure 2. `List` **computation tree for** $N = 3$.

A *generator* is a concise representation of these simple rules observed during the execution of the program on small inputs (e.g. $N = 1, 2, 3$). In Section 5, we formally define a class of *generators*, called *branch policy generators*, that succinctly encodes such rules.

In the third stage of our algorithm, we use the above set of rules (i.e. the generator) to restrict symbolic execution to a small subset of feasible execution paths of the program on inputs of larger sizes. Specifically, as our algorithm uses symbolic execution to explore the computation tree of the example program for larger input sizes, it prunes the search by never taking the "false" branch of the conditional at line 12 when it is possible to take the "true" branch instead.

Thus, the symbolic test generation produces only test inputs in which each new element is inserted at the end of the sorted linked list. Only a single feasible execution path will be considered for each input size N, rather than all $N!$ feasible paths. Moreover, it is easy to see that this is the execution path exhibiting the worst-case complexity of the example program. An input corresponding to the worst-case execution path is then computed using constraint solving. For example, if $N = 4$, then $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, $x_4 = 4$, exhibits the worst-case complexity.

3. Background

In this section, we briefly describe the programming model and test generation techniques WISE uses.

3.1. Programming Model

We describe our technique on a simple imperative programming language. A program P in this language consists of a sequence of labeled statements. The statements are one

of: (1) an input statement $m := \text{INPUT}()$, (2) an assignment $m := e$ to lvalue m of the value of e, an expression free of side effects, (3) a conditional `if` p `then goto` l, where l is the label of another statement in the same function and p is a predicate free of side effects, or (4) an `Exit` statement, terminating the program.

For statement s, let $\text{succ}(s)$ denote the set of statements which can immediately follow s. Note all statements have one successor except for conditionals, which have two successor statements, and the `Exit` statement, which has none.

We say that a program is *run on an input $x_1, \ldots, x_n$ of size n*, if the program executes the input statement exactly n times and receives the inputs $x_1, \ldots, x_n$ in order.

Running a program produces an *execution path*—a sequence $\langle s_1, \ldots, s_k \rangle$ of labeled statements where s_1 is the unique initial statement of P and where $s_{i+1} \in \text{succ}(s_i)$ for each s_i. We consider only programs which always terminate when run on an input of any fixed size.

3.2. Symbolic Execution

Let $s_1, \ldots, s_k$ be a path through program P. Symbolic execution [6, 20] is a procedure for computing a *symbolic path constraint* $\Phi(P, n, \langle s_1, \ldots, s_k \rangle)$, a formula over program inputs $x_1, \ldots, x_n$ that exactly characterizes the inputs which cause program P to execute along path $s_1, \ldots, s_k$. That is, the first k statements program P will execute on concrete inputs $x_1, \ldots, x_n$ are exactly $s_1, \ldots, s_k$ if and only if $x_1, \ldots, x_n$ satisfy the formula $\Phi(P, n, \langle s_1, \ldots, s_k \rangle)$.

We omit the details of any particular procedure for symbolic execution or for solving the resulting symbolic path constraints. WISE can be applied to any imperative, Java-like language for which we have a complete symbolic execution and decision procedure.

3.3. Symbolic Test Generation

We now present in Algorithm 1 a standard symbolic or concolic test generation procedure (e.g. [11, 25, 5]) leveraging the symbolic execution procedure Φ from the previous section. Function *SymbolicTestGen*$(P, n, \langle s_1, \ldots, s_k \rangle)$ takes three arguments: a program P, an input size n, and a partial execution path—that is, a sequence $\langle s_1, \ldots, s_k \rangle$ of statements of P. When called on some P, n, and $\langle s_1, \ldots, s_k \rangle$, the procedure returns the set of all feasible execution paths on size-n inputs that have $s_1, \ldots, s_k$ as a prefix. Further, for each such path, an input $x_1, \ldots, x_n$ is returned that realizes the path.

Thus, *SymbolicTestGen*$(P, n, \langle s_1 \rangle)$, where s_1 is the first statement of program P, will return every possible path through P (on a size-n input), as well as test cases $x_1, \ldots, x_n$ exercising each path.

Algorithm *SymbolicTestGen*$(P, n, \langle s_1, \ldots, s_k \rangle)$ works by finding all feasible single-statement extensions $\langle s_1, \ldots, s_k, s \rangle$ of the partial execution $\langle s_1, \ldots, s_k \rangle$,

Algorithm 1 *SymbolicTestGen*$(P, n, \langle s_1, \ldots, s_k \rangle)$

  if $s_k ==$ `Exit` **then**
    $x_1, \ldots, x_n \leftarrow$ an input satisfying $\Phi(P, n, \langle s_1, \ldots, s_k \rangle)$
    return singleton set $\{(x_1, \ldots, x_n; \langle s_1, \ldots, s_k \rangle)\}$
  else
    $\Gamma \leftarrow$ empty set
    for $s \in \text{succ}(s_k)$ **do**
      if $\Phi(P, n, \langle s_1, \ldots, s_k, s \rangle)$ is satisfiable **then**
        $\Gamma \leftarrow \Gamma \cup$ *SymbolicTestGen*$(P, n, \langle s_1, \ldots, s_k, s \rangle)$
      end if
    end for
    return Γ
  end if

Algorithm 2 WISE(program P, N, M)

  // Exhaustive test generation for small inputs.
  for $i = 1$ to N **do**
    $tests_i \leftarrow$ *SymbolicTestGen*$(P, i, \langle s_1 \rangle)$
  end for
  // Search for a *worst-case* generator.
  $G \leftarrow$ *FindGoodGenerator*$(tests_1, \ldots, tests_N)$
  // Guided test generation for larger inputs.
  for $i = 1$ to M **do**
    $tests_i \leftarrow$ *GuidedTestGen*$(P, i, G, \langle s_1 \rangle)$
    output worst-case input in $tests_i$
  end for

and then recursing on each such extended execution. Specifically, for each $s \in \text{succ}(s_k)$ it symbolically executes $\langle s_1, \ldots, s_k, s \rangle$, producing path constraint $\Phi(P, n, \langle s_1, \ldots, s_k, s \rangle)$. It checks the feasibility of the extension by checking if the path constraint is satisfiable (by, e.g., using an off-the-shelf SMT solver). The recursion stops whenever a path reaches the end of program P—i.e. an `Exit` statement.

The feasible execution paths of a program P can be seen as forming a binary tree, called the *computation tree* of P. Each node in the tree is labeled with a statement from P, with s_1 at the root of the tree. Letting $s_1, \ldots, s_k$ be the path from the root of the tree to some vertex labeled with s_k, the children of s_k correspond to the statements s for which $s_1, \ldots, s_k, s$ are prefixes of feasible execution paths—i.e. for which $\Phi_n(s_1, \ldots, s_k, s)$ is satisfiable. Thus, each root-to-leaf path in the tree corresponds to a feasible execution. Algorithm 1 can be viewed as a depth-first search on the *computation tree* of a program P.

4. Algorithm

For a given $n > 0$ and a program P, we say that a feasible execution path is a *worst-case execution path* if it contains the maximum number of conditional statements contained by any feasible execution of P on inputs of size n. An input that results in a worst-case execution path is called a *worst-case input*. In this section, we describe in detail the WISE algorithm, a complexity testing algorithm for finding a worst-case input for input sizes up to a given M.

4.1. The WISE Algorithm

We described a naïve complexity testing algorithm in Section 1. The algorithm symbolically enumerates all feasible execution paths of the program for each input size from 1 to M, selects a single worst-case execution path for each input size, and uses constraint solving to generate a worst-case input for that path. Although this naïve algorithm is correct, it fails to scale to larger input sizes because

the number of feasible execution paths often increases exponentially as the input size is increased. We propose the WISE algorithm to attempt to significantly reduce this scalability problem. Formally, the WISE algorithm, given as Algorithm 2, takes as parameters two natural numbers N and M, and a program P, which can be run on input $x_1, \ldots, x_n$ of size n for any $n > 0$. The algorithm works in the following three stages:

1. First, WISE uses symbolic execution to generate test inputs for all feasible execution paths of the program when run on $N = 1, 2,$ or 3.

2. Second, from these execution paths, WISE extracts a *generator G* (which we will define later), an entity that characterizes a small subset of execution paths including a worst-case execution path of the program.

3. Finally, WISE uses *guided* symbolic test generation to produce test inputs of larger sizes only for the paths conforming to the generator G. For each larger input size, WISE outputs a worst-case generated test input.

4.2. Generators

Before proceeding, let us provide a generic definition of a *generator*. A *generator* for a program P is a function mapping every prefix of an execution path of P to either **true** or **false**. We require that generators be *consistent*, meaning that whenever $G(\sigma)$ is true, $G(\sigma')$ is also true for every prefix σ' of σ. We say that a generator G *generates* an execution path or path prefix σ, or that σ *conforms* to a G, if $G(\sigma)$ is true.

We say that a generator G is a *worst-case generator* if there exists a sequence $\sigma_1, \sigma_2, \sigma_3, \ldots$ such that, for each σ_n, both $G(\sigma_n)$ is true and σ_n is a worst-case execution path for inputs of size n. In the WISE algorithm, the requirement that the second stage computes a worst-case generator guarantees that the third stage, which produces only execution paths conforming to this generator, will still produce worst-case test inputs.

Note that the definition of a generator is generic in the sense that we do not specify how we create a such function. A *trivial* example of a worst-case generator is the one that returns true for execution paths. Note that the second stage of the above algorithm could return this generator; however, it would not help to prune the search space in the third stage.

Therefore, our goal should be to come with a good worst-case generator—one that generates only a small set of feasible execution paths. The generator described in Section 2 is an ideal generator, as it generated only the single longest path for each input size.

4.3. Guided Symbolic Test Generation

In Section 3.2 we gave an algorithm *SymbolicTestGen* for exhaustive generation of test inputs of size n. In the next section, we present a modification of that algorithm, *GuidedTestGen*, which uses a generator G to produce test inputs more efficiently by pruning the computation tree of program P, while still producing a worst-case input for each input size n.

An invocation of *GuidedTestGen*$(P, n, G, \langle s_1 \rangle)$ recursively finds all feasible execution paths among those *that conform to generator G*. Furthermore, inputs are found that realize each such execution path. Notice that Algorithm 3 is identical to *SymbolicTestGen* in Algorithm 1, except that recursive calls are guarded by a check that the path to be explored conforms to G.

Thus, we can view *GuidedTestGen* as a depth-first search of the computation tree of the program under test, except that we prune as early as possible any subtree containing no generated execution paths.

This pruning is critical to the feasibility of the WISE algorithm. For a typical program, the size of the computation tree grows at least exponentially with the input size, rendering exhaustive test generation completely intractable. But with an appropriately chosen generators, guided symbolic symbolic test generation can often prune all but a polynomially-sized portion of the computation tree, enabling worst-case test inputs to be efficiently found.

5. Approximate Worst-Case Generators for the WISE Algorithm

We have described the WISE algorithm without a detailed description of the sub-algorithm *FindGoodGenerator*. The description of this algorithm forms the content of this section. Note that we are free to choose any *worst-case* generator—the WISE algorithm will output a worst-case test input for each input size as long as the generator yields a worst-case execution path for each input size. We have previously noted that the trivial generator which generates all execution paths is worst-case. This trivial generator, however, does not prune the search for larger worst-case

---

Algorithm 3 *GuidedTestGen*$(P, n, G, \langle s_1, \ldots, s_k \rangle)$

if $s_k ==$ `Exit` **then**
  $x_1, \ldots, x_N \leftarrow$ input satisfying $\Phi(P, N, \langle s_1, \ldots, s_k \rangle)$
  return singleton set $\{(x_1, \ldots, x_N; \langle s_1, \ldots, s_k \rangle)\}$
else
  $\Gamma \leftarrow$ empty set
  for $s \in \text{succ}(s_k)$ **do**
    if $G(\langle s_1, \ldots, s_k, s \rangle)$ **then**
      if $\Phi(P, n, \langle s_1, \ldots, s_k, s \rangle)$ is satisfiable **then**
        $\Gamma \leftarrow \Gamma \cup$ *GuidedTestGen*$(P, n, G, \langle s_1, \ldots, s_k, s \rangle)$
      end if
    end if
  end for
  return Γ
end if

---

inputs, and thus the WISE algorithm provides no benefit if such a generator is returned. Our goal should be to find generators that are worst-case but generate only a small subset of feasible execution paths.

In this paper, we propose a class of generators, called *branch policies*, which we have found in practice to be quite effective in efficiently finding worst-case inputs. Moreover, we give an efficient, approximate *FindGoodGenerators* algorithm to select a branch policy given an exhaustive list of execution paths for small input sizes.

The given *FindGoodGenerators* algorithm is approximate in two senses: First, it does not guarantee that the generator it returns is a *worst-case* generator. In Section 6, however, we show that, for any given program P, the approximate *FindGoodGenerators*$(tests_1, \ldots, tests_N)$ will return a worst-case generator for sufficiently large N. (And we observed in our experiments that fairly small N were sufficient.) Second, of the possible worst-case generators *FindGoodGenerators* could return, we do not guarantee that an optimal generator is found—i.e. one that best prunes the space of feasible execution paths. Rather, *FindGoodGenerator* uses a greedy heuristic search.

5.1. Branch Policy Generators

In this section, we describe *branch policies*, a class of generators that characterize execution paths by specifying which branches they may and may not contain.

The motivation for branch policy generators is the observation that some worst-case execution paths always follow a particular pattern of branches, i.e. they avoid some branches completely or prefer one branch over the other when a choice is available. For example, in the sorted linked list in Figure 1, a worst-case execution path prefers to take the "true" branch of the conditional at line 12 over the "false" branch of the same conditional. Similarly, in case of Quicksort, we obtain a worst-case execution path if the array to be sorted is partitioned into arrays of size 1

and $(n-2)$ in each invocation. This occurs if the "false" branch of a certain conditional in the quicksort algorithm is taken whenever it is feasible. We observed similar patterns in many standard algorithms. Branch policies capture this pattern by both explicitly disallowing some true or false branches and by marking other branches as allowed only when the alternative is infeasible.

Specifically, a *branch policy* BP divides the true and false branches of every static conditional statement in the program under test into three categories: (1) forbidden branches, (2) branches permitted only if *every* input reaching the branch causes the branch to be taken, and (3) fully permitted branches. More formally, a branch policy $BP: S_c \times S \to \{0, 1/2, 1\}$ is a function, where S_c is the set of all conditional statements in the program and S is the set of all statements in the program. We say that the edge from s to s' in an execution path is forbidden if $BP(s, s')$ is 0. Similarly, that the edge from s to s' in an execution path is permitted if $BP(s, s')$ is 1. If $BP(s, s')$ is $1/2$, then we say that the edge from s to s' is permitted after some path $s_1, \ldots, s_k = s$ only when every input yielding partial execution path $s_1, \ldots, s_k = s$ also yields $s_1, \ldots, s_k = s, s'$.

5.2. Test Generation with Branch Policies

Given a branch policy BP, we can define a generator G_{BP} as follows. $G(s_1, \ldots, s_k)$ is true iff, for each conditional statement s_i on the path with $i < k$, either:

- $BP(s_i, s_{i+1})$ is 1, or

- $BP(s_i, s_{i+1})$ is $1/2$ and if any input satisfies $\Phi_N(s_1, \ldots, s_i)$, then the input also satisfies $\Phi_N(s_1, \ldots, s_i, s_{i+1})$. (I.e. any alternative extension of $s_1, \ldots, s_i$ other than $s_1, \ldots, s_i, s_{i+1}$ is infeasible.)

Once we have the generator G_{BP}, guided symbolic test generation can be done using Algorithm 3.

5.3. Finding a Good Branch Policy

In this section, we give a precise specification for a *FindGoodGenerator* implementation for branch policies and describe a simple, naïve implementation. Further, we describe an optimized, greedy implementation which can be used to select a branch policies in practice.

In selecting a branch policy, we have two competing goals: the branch policy should generate as few execution paths as possible so that guided test generation is efficient, but the branch policy must still generate worst-case executions for every input size. We balance these goals by giving a *FindGoodGenerator*($tests_1, \ldots, tests_N$) algorithm which will only return branch policies which generate a worst-case execution path for each input size from 1 to N. But, among these potential generators, we return one which generates the least number of total execution paths on N inputs.

Formally, let $C_n(G_{BP})$ denote the number of execution paths on inputs of size n generated by G_{BP}. Further, let max_n denote the set of all branch policies which generate a worst-case execution on inputs of size n, and $max = \bigcap_{i=1}^{N} max_i$. Then, procedure *FindGoodGenerator*, for executions $tests_1, \ldots, tests_N$, returns:

$$\arg\min_{G_{BP} \in max} C_N(G_{BP})$$

Note that we count only the executions on inputs of size N, rather than on inputs of sizes up to N, on the theory that this provides a better estimate of the relative costs of different branch policies for larger input sizes.

Naïve Selection Algorithm. A naïve algorithm for computing such a G_{BP} is to simply enumerate all possible branch policies for a program P—there are only finitely many because P contains only finitely many statements. For each potential G_{BP}, we pass through the exhaustive list $tests_i$ of i-input execution paths to check whether G_{BP} generates a worst-case execution for each input size, as well as to count the total number of N-input executions it generates. Then, we simply return any acceptable G_{BP} with a minimal count. Note that this algorithm will always return some branch policy, because the trivial policy $\top$, which maps each branch to 1, clearly generates a worst-case execution path for each input size.

Optimization 1: Summarizing Executions. There is a natural partial order on any class of generators—we can define $G_1 \leq G_2$ iff G_2 generates every path that G_1 generates. This ordering can be efficiently checked for two branch policies G_{BP} and $G_{BP'}$ by observing that $G_{BP} \leq G_{BP'}$ iff $BP(s, s') \leq BP'(s, s')$ for every pair $(s, s') \in S_c \times S$.

Further, for every n-input execution path or path prefix $\sigma = s_1, \ldots, s_k$ there is a *least* branch policy BP_σ such that $G_{BP_\sigma}(\sigma)$ is true. In particular,

- $BP_\sigma(s, s') = 1$ if there is some i such that $s_i = s$, $s_{i+1} = s'$, and $\Phi_n(s_1, \ldots, s_i) \not\Rightarrow \Phi_n(s_1, \ldots, s_{i+1})$.

- $BP_\sigma(s, s') = 0$ if s, s' never appears in σ.

- $BP_\sigma(s, s') = 1/2$ otherwise.

These two allow us to work with a summarization of the executions $tests_1, \ldots, tests_N$. Specifically, we need only max_i, the set of *least* branch policies BP_σ for each worst-case execution σ on i inputs, and $count_i(BP)$, a table recording for each branch policy BP the number of executions σ on i inputs for which G_{BP} is the least generator.

With this summarization, we can restate the naïve algorithm as computing:

$$\arg\min_{G_{BP}} \sum_{G_{BP'} \leq G_{BP}} count_N(BP')$$

6

Algorithm 4 *FindGoodGenerator*($tests_1, \ldots, tests_N$)

$G \leftarrow \bot$
for $i = N$ to 1 **do**
  $(max_i,\ count_i) \leftarrow$ *SummarizeExecutions*($tests_i$)
  $G' \leftarrow \arg\min_{G' \in max_i} \sum_{H \leq G \sqcup G'} count_N(H)$
  $G \leftarrow G \sqcup G'$
end for
return G

These summaries can be computed with a single pass through each of $tests_i$. In practice, we interleave the computation of the summaries with the exhaustive test generation, and thus do not explicitly store any $tests_i$. This is a big win, as the number of distinct *least* branch policies is typically much smaller than the number of execution paths.

Optimization 2: Greedy (Approximate) Search. There is also a natural semilattice structure on any class of generators, where the join $G_1 \sqcup G_2$ is the least generator such that $G_1, G_2 \leq G_1 \sqcup G_2$. For branch policies, this join is easy to compute: $(G_{BP_1} \sqcup G_{BP_2})(s, s') = G_{BP}$, where for all statements s and s', we have $BP(s, s') = \max(BP_1(s, s'), BP_2(s, s'))$.

Algorithm 4, $FindGoodGenerator$, greedily selects a branch policy using summaries of the test executions for each input size from 1 to N. In the first iteration of the main loop in $FindGoodGenerator$, a branch policy will be found which generates a worst-case execution on N inputs and also generates the least number of total N-input executions. For many programs, this branch policy will also generate a worst-case path for input sizes 1 to $N - 1$, in which case later iterations of the loop will have no effect and the branch policy will be returned.

However, for some programs this N-input branch policy G may not generate any worst-case execution paths for some smaller input size k. (Consider a program which runs one procedure for even-size inputs and one for odd-size inputs.) At iteration $i = k$, the loop in $FindGoodGenerator$ finds the branch policy G' which generates a worst-case k-input path, such that the combined branch policy $G \sqcup G'$ generates the minimal number of test executions on N inputs. Thus, the final branch policy is guaranteed to generate a worst-case path for each input size up to N.

Note that this is only a greedy approximation of the absolute "minimal" generator returned by the naïve algorithm. However, this approximation did not seem to negatively impact the scalability for the few benchmarks in which branch policies for different input sizes needed to be merged.

This greedy $FindGoodGenerators$ has to loop over only the branch policies for each input size i which are the *least* branch policies for worst-case executions. In practice, this is much more efficient than enumerating all possible branch policies, as there are typically only a handful of worst-case executions to consider.

6. Theoretical Guarantees for WISE

In the WISE algorithm, we pick some N and search for a generator G which generates a worst-case path for each input size from 1 to N. Then, when searching for larger worst-case inputs, we restrict ourselves only to execution paths conforming to G. We would like some assurance that our generator selection will lead us to find the actual worst-case paths for larger input sizes. We show below that, when restricted to a finite class of generators, for every program P there is an N for which the WISE algorithm is guaranteed to produce worst-case inputs for larger input sizes.

Proposition 6.1. *Let P be a program and $\mathcal{G}$ be a finite set of generators for P. Then there exists an $N > 0$ such that if $G \in \mathcal{G}$ generates a worst-case path for each input size from 1 to N, then G is a worst-case generator.*

Proof. Let $\mathcal{G}_i \subseteq \mathcal{G}$ be the set of all generators which generate a worst-case path for each input size from 1 to i. Clearly $\mathcal{G}_i \supseteq \mathcal{G}_{i+1}$ for each i.

Consider the descending chain $\mathcal{G}_1 \supseteq \mathcal{G}_2 \supseteq \cdots$. Because $\mathcal{G}$ is finite, this chain must become stationary at some $\mathcal{G}_N = \mathcal{G}_{N+1}$. Then, any $G \in \mathcal{G}_N$ is a worst-case generator.

In fact, $\mathcal{G}_N$ contains all worst-case generators in $\mathcal{G}$. $\qquad\square$

In particular, the set of possible branch policies for a program P is finite because P contains only finitely-many conditional branches. The second stage of the WISE algorithm computes a branch policy that generates a worst-case path for input sizes $1, \ldots, N$. Thus, if WISE is run with a sufficiently-large N, this policy is a worst-case generator.

Other examples of finite classes of generators include context-sensitive branch policies, in which, e.g., the policy for some conditional statement can depend on the previous k conditional branches or the nearest k enclosing function calls, or bounded-size regular expressions or context-free grammars over the statements of P.

It is easy to see that Proposition 6.1 does not hold for some infinite classes of generators. For example, consider the class of generators consisting of all regular expressions over the alphabet of statements in a program P. For any N, consider a generator $G = \sigma_1 | \sigma_2 | \cdots | \sigma_N$, where each σ_i is a worst-case execution path for inputs of size i. Although G generates a worst-case path for each input size from 1 to N, it generates no paths for any larger input size.

7. Experimental Results

We have implemented WISE in a prototype tool for Java. We use concolic execution [25], a combination of concrete and symbolic execution, to explore the feasible execution paths of a program. We have applied WISE to a number of data structure and algorithms. We obtained several of these

Benchmark	Statistic		Input Size (N)									N*
			1	2	3	4	5	10	15	20	30	
Sorted Linked-List insert	Exhaustive	Paths	1	2	6	24	120	3628800	-	-	-	2
	Guided	Paths	1	1	1	1	1	1	1	1	1	
		Iterations	1	2	3	4	5	10	15	20	30	
	Path-Length	Longest	2	3	4	5	6	11	16	21	31	
		Average	2.00	2.50	3.00	3.50	3.99	6.48	8.98	11.48	16.48	
Heap insert (JDK 1.5)	Exhaustive	Paths	1	2	4	12	36	20736	21233664	-	-	2
	Guided	Paths	1	1	1	1	1	1	1	1	1	
		Iterations	1	2	3	5	7	20	35	55	95	
	Path-Length	Longest	7	13	20	29	38	89	144	209	339	
		Average	4.00	5.50	5.50	6.22	6.25	6.60	6.73	6.75	6.88	
Red-Black Tree search (3rd Party)	Exhaustive	Paths	3	10	42	216	1320	-	-	-	-	8
	Guided	Paths	1	1	1	1	1	1	1	1	1	
		Iterations	3	5	5	7	7	11	13	13	17	
	Path-Length	Longest	4	7	7	10	10	16	19	19	25	
		Average	4.00	6.01	7.00	8.20	9.00	11.66	13.41	14.63	16.34	
Quicksort (JDK 1.5)	Exhaustive	Paths	1	2	6	24	120	-	-	-	-	8†
	Guided	Paths	1	1	1	1	1	1	1	1	1	
		Iterations	1	2	4	7	11	18	21	23	28	
	Path-Length	Longest	5	10	17	26	37	144	309	484	974	
		Average	4.00	7.50	12.15	17.92	24.71	97.12	174.82	260.40	447.62	
Binary Search Tree search	Exhaustive	Paths	1	3	13	75	541	102247562	-	-	-	3
	Guided	Paths	1	1	1	1	1	1	1	1	1	
		Iterations	1	3	6	10	15	55	120	210	465	
	Path-Length	Longest	1	4	7	10	13	28	43	58	88	
		Average	1.00	4.00	6.00	7.49	8.70	12.58	14.88	16.63	18.97	
Mergesort (JDK 1.5)	Exhaustive	Paths	1	2	6	24	120	3628800	-	-	-	7
	Guided	Paths	1	1	1	1	1	125	2216426	-	-	
		Iterations	1	2	4	7	11	271	4157281	-	-	
	Path-Length	Longest	4	8	14	22	32	106	206	-	-	
		Average	4.00	7.50	11.75	16.45	21.45	63.57	123.87	-	-	
Bellman-Ford	Exhaustive	Paths	1	2	63	184875	-	-	-	-	-	2
	Guided	Paths	1	1	1	1	1	1	1	1	1	
		Iterations	1	2	5	9	14	54	119	209	464	
	Path-Length	Longest	1	55	147	315	583	4263	14043	32923	109983	
		Average	1.00	54.50	143.10	314.25	582.97	4263.00	14043.00	32923.00	109983	
Dijkstra's	Exhaustive	Paths	1	1	4	56	2592	-	-	-	-	3
	Guided	Paths	1	1	1	1	1	1	1	1	1	
		Iterations	1	1	1	1	1	1	1	1	1	
	Path-Length	Longest	12	32	62	102	152	552	1202	2102	4652	
		Average	12.00	32.00	62.00	102.00	152.00	552.00	1202.00	2102.00	4652.00	
Traveling Salesman	Exhaustive	Paths	1	1	3	297	-	-	-	-	-	5†
	Guided	Paths	1	1	1	1	1	-	-	-	-	
		Iterations	1	1	3	6	10	-	-	-	-	
	Path-Length	Longest	4	12	35	127	587	-	-	-	-	
		Average	4.00	12.00	34.62	118.09	451.70	288291.52	∼8.66e7	-	-	

Table 1. Results of exhaustive and guided symbolic test generation for various benchmarks and input sizes. For each benchmark and input size, we give five quantities: *Exhaustive* **reports the total number of feasible paths explored by exhaustive symbolic test generation.** *Guided: Paths* **and** *Guided: Iterations* **report the number of explored paths and calls to the constraint solver during guided symbolic test generation, using the branch policy found by** WISE **on inputs of size up to** N^***.** *Path-Length: Longest* **reports the length of longest path found during guided test generation.** *Path-Length: Average* **reports the average path length observed over 10,000 executions of the benchmark program on random inputs of the given size. An '-' entry indicates the run could not be completed in 3 hours.**

benchmark programs from Sun's JDK 1.5 and implemented the rest on our own.

We know from Section 6 that, for any program, if we run WISE with sufficiently large N then we will produce a worst-case branch policy—i.e. a branch policy that generates worst-case execution for larger input sizes. In our experiments, we aim to validate the following hypotheses:

1. In practice, the necessary N is small enough to make WISE feasible—that is, WISE's exhaustive test generation is feasible for inputs of size up to this N.

2. The produced worst-case branch policies significantly prune the search space for finding larger worst-case executions.

Thus, for each of our benchmarks, we run the first two stages of WISE—exhaustive test generation and computation of a branch policy—for as large N as was feasible in three hours on our test machine.

We then determined by hand the minimal N for which a worst-case generator is found—denote this by N^*—and then ran the third stage of WISE with the worst-case branch policy found for $N = N^*$ on inputs of size up to 30. The results are summarized in Table 1.

Hypothesis 1. Table 1 shows that, for all of our benchmarks, the sufficiently-large N^* was less than nine. Further, for all but the Traveling Salesman benchmark, exhaustive test generation was very feasible on inputs of size up to the found N^*, requiring no more than a few minutes of computation.

(The number and length of execution paths grow so rapidly in the Traveling Salesman benchmark, a branch-and-bound search with a trivial bounding procedure, that exhaustive test generation was not feasible even for a complete graph with $N = 5$ vertices. However, the output of WISE for $N = 4$ contained a worst-case branch policy and allowed us to determine that $N^* = 5$ for this benchmark.)

Hypothesis 2. Table 1 shows that, for all but the Mergesort benchmark, the worst-case branch policies computed by WISE generate only a single path for larger input sizes. That is, in these cases WISE finds worst-case generators that prune away the entire search space except for a single worst-case execution path. Thus, WISE's guided test generation is very efficient, producing worst-case executions for input sizes of up to 30 (and beyond) in only a few seconds and with a small number of constraint solver calls.

The two exceptions are the Traveling Salesman and Mergesort benchmarks. As mentioned above, the worst-case execution paths for the Traveling Salesman benchmark rapidly become extremely long, as the backtracking search must explore all $(N-1)!$ cycles in the N-vertex input graph. (For $N = 9$, the longest path contains 1.5M branches.)

Thus, even though our branch policy generates only a single path, guided test generation is infeasible for $N \geq 10$.

In the Mergesort benchmark, on the other hand, our worst-case branch policy is not effective enough in pruning the search for worst-case executions. There is a conditional statement at which Mergesort selects the least item from one of the two lists it is merging. Although Mergesort always has the same asymptotic complexity, the worst-case empirical complexity is achieved when Mergesort alternates between the two lists during merging. Our branch policy merely records that a worst-case path could take either branch of this conditional. Thus, although WISE's guided test generation greatly prunes the search space (e.g. by a factor of 600,000 for $N = 15$), it must still explore exponentially many paths.

Observations. One can gain insight into the asymptotic worst-case complexity of an algorithm by examining the empirical complexity of the worst-case inputs generated by WISE. (For example, by plotting the path lengths versus input size.) Further, the worst-case branch policy generator found by WISE can provide insight into the source of an algorithm's worst-case complexity. Here we highlight some of the interesting branch policies and empirical complexities we observed in our experiments.

For each of our four data structure benchmarks, we wrote a driver program which inserts N integers into the data structure and then used WISE to find the worst-case complexity *of one search or additional insertion*. Table 1 shows that, for insertion into a sorted linked-list and look-ups in a red-black tree, the worst-case executions found by WISE are only 1.5 to 2 times longer than our observed average path lengths, as these methods have the same average-case and worst-case asymptotic complexity—$O(n)$ and $O(\log n)$, respectively. Similarly, for insertions into a binary heap and look-ups in a binary search tree, the WISE's executions show the $O(\log n)$ and $O(n)$ worst-case complexities, even though the average-case complexities are, respectively, nearly constant and $O(\log n)$. For the list, heap, and binary tree, the branch policies neatly constrain the inputs to be an increasing or decreasing sequence, by requiring allowing only the "true" or "false" branch of conditionals comparing newly-inserted elements to previous ones.

Similarly, WISE highlights the $O(n^2)$ worst-case of Quicksort, while the average-case complexity is only $O(n \log n)$. Our branch policy requires that, whenever feasible, each element must be less than the pivot when compared. Thus, as the JDK 1.5 Quicksort uses a median-of-3 pivot, each split leaves a sub-array with $(n - 2)$ elements. (Note: For $n > 40$, the JDK 1.5 Quicksort actually switches to a median-of-9 for the pivot. We remove this from our benchmark, because it would force WISE to do exhaustive test generation up to $N^* = 40$.)

For our three graph algorithm benchmarks, we use com-

plete graphs on n vertices as inputs. For Bellman-Ford and Dijsktra's Algorithm, WISE produces inputs exhibiting the $O(n^3)$ and $O(n^2)$ worst-case complexity. Roughly, the branch policies for these benchmarks require that we relax an edge whenever feasible. For Bellman-Ford, this ensures that there is a negative-weight cycle so that the algorithm does not terminate early. For our Traveling Salesman benchmark, although WISE is only able to generate worst-case inputs up to $n = 9$, these inputs suggest that the worst-case complexity is roughly $O(n!)$. WISE's branch policy requires that, whenever feasible, the branch-and-bound search not prune its current subtree. (It prunes a path whenever its length exceeds that of the shortest cycle found so far.)

8. Limitations

Our experimental results demonstrate that our complexity testing technique WISE can effectively find large worst-case inputs for many standard algorithms and data structure operations. However, it is not clear whether WISE can be applied to larger applications with multiple components.

Such large applications can pose several challenges to WISE: (1) They may be too complex to be handled by existing symbolic execution and constraint solving techniques. (2) It mot not be feasible for WISE to perform exhaustive test generation for large enough input sizes, both because a larger input size may be required to find a generator and because of an increased number of paths even for small inputs. For example, to produce a generator for an application that uses one algorithm for inputs of size up to 40 and another for larger inputs, WISE would need to perform exhaustive test generation for inputs of size up to at least 41, which is likely to be intractable. (3) The generators found by WISE may not prune enough executions for larger input sizes. For example, our Mergesort branch policy still leaves an exponential search for worst-case executions. This occurs because a worst-case Mergesort execution must alternate between the two sides of a critical conditional, but our generator can only capture that worst-case paths are always permitted to take either branch.

More sophisticated generators could help in addressing these challenges in scaling WISE up to larger applications. For example, generators that encode that certain branches must be taken in alternating or other regular sequences. Further, it may be possible for a technique to produce more and more precise generators as it generates and explores executions for larger and larger input sizes.

9. Related Work

Loop bounds in iterative programs have traditionally been calculated using *ranking functions* [8]. Ranking functions are bounded functions on loop variables that keep on decreasing in every loop iteration. A bound on the ranking functions gives an approximation of the loop bound.

Gulavani and Gulwani [14] used this observation to develop static analysis based methods to construct numeric abstract domains for timing analysis. The execution time obtained by these static analysis techniques is an upper bound on the actual execution time of the program. Their approach produces an upper bound on the worst-case computational complexity, whereas WISE produces a lower bound on the worst-case computational complexity.

Profilers [13, 27, 1] based on dynamic program analysis are often used to find performance bottlenecks in programs. These tools periodically or continuously sample the program counters and collect various timing statistics about a program. More recently, Goldsmith et al. [12] have come up with a novel idea to empirically compute the observed computation complexity. Their technique uses profiled data in conjunction with curve-fitting to compute empirical computational complexity. The complexity bound that they compute is based on real-world usage and may not represent the worst-case complexity. WISE directs test generation so that it can discover the worst case complexity.

There has been a large body of work in estimating the worst case execution time for embedded and real-time systems [22, 30, 16, 17, 18]. The major concern for these systems is modeling the underlying architecture and environment such as cache hit/miss, instruction pipelining etc. This problem is tackled using several techniques from pattern matching and abstract interpretation (e.g. interval based abstract interpretation [16]). An exciting piece of recent work in this area has been game theoretic timing analysis [26]. Our approach is orthogonal to these efforts. Further, most of these techniques assume that loops have finite bounds, independent of input size, and they completely unroll them. We handle loops with bounds depending on the input.

10. Conclusion

We introduce complexity testing, a novel automated test generation technique that finds worst-case inputs for a program unit. Experiments on several standard benchmark programs show that complexity testing can efficiently discover worst-case executions for large input sizes.

Complexity testing is a first step towards automated testing for performance and scalability. We believe that complexity testing can help relieve the burden of manual computational complexity analysis, similar to the way that automated software testing and software model checking help relieve the burden of manual program verification.

Acknowledgment

References

[1] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP 2004 - Object-Oriented Programming*, volume 3086 of *LNCS*, pages 170–194. Springer, 2004.

[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In *Proc. of the 26th ICSE*, pages 326–335, 2004.

[3] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Software Testing and Analysis*, pages 123–133, 2002.

[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS 2006)*, 2006.

[6] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2:215–222, 1976.

[7] L. A. Clarke and D. J. Richardson. Symbolic evaluation – an aid to testing and verification. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 141–166, New York, NY, USA, 1984. Elsevier North-Holland, Inc.

[8] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS 2001: Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–81. Springer-Verlag, 2001.

[9] P. T. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application*, pages 363–382. ACM, 2006.

[10] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, 2000.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[12] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Foundations of software engineering (ESEC-FSE '07)*, pages 395–404. ACM, 2007.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.

[14] B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstractionand max operatorwith application in timing analysis. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2008.

[15] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proc. the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 231–244, 1998.

[16] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 57–66. IEEE Computer Society, 2006.

[17] N. Halbwachs, Y. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[18] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. V. Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Syst.*, 18(2-3):129–156, 2000.

[19] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th Int. Conf. on TACAS*, pages 553–568, 2003.

[20] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[21] B. Korel. A dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, November 1990.

[22] Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwar Academic Publishers, 1999.

[23] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[24] J. Offut and J. Hayes. A Semantic Model of Program Faults. In *Proc. of ISSTA'96*, pages 195–200, 1996.

[25] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.

[26] S. Seshia and A. Rakhlin. Game theoretic timing analysis. In *International Conference on Computer-Aided Design 2008*, 2008.

[27] G. Sevitsky, W. de Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *TOOLS '01: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 85. IEEE Computer Society, 2001.

[28] N. Tillmann and W. Schulte. Parameterized unit tests. In *Foundations of Software Engineering (ESEC/FSE)*, pages 253–262, 2005.

[29] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[30] R. Wilhelm. Determining bounds on execution times. In *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.