Automated Test Data Generation with SAT

Robert Seater and Gregory Dennis
{rseater, gdennis}@mit.edu

ABSTRACT

We present a novel technique for automatically generating a suite of test inputs to an object-oriented procedure. The suite is guaranteed to kill all mutant procedures produced from a given catalog of mutation operators, so long as those mutants could be detected by some test within user-provided bounds.

Our test input generator constructs a *mutation-parameterized* version of the procedure, whose behavior may differ from the original by a single mutation. It encodes the original and mutation-parameterized procedures in a first-order relational formula, a solution to which includes both a mutation and a test input that detects it. Our tool iteratively invokes a constraint-solver to find such tests and adds them to the test suite. After each iteration, it amends the formula to ensure the next mutant found was not killed by a previous input. This process is repeated until the formula becomes unsatisfiable, at which point all mutants have been detected.

We evaluate an implementation of this technique on a series of small benchmarks.

1. INTRODUCTION

The degree to which a test suite explores the behavior of a program is the suite's *coverage*. For a well chosen coverage metric, the greater the coverage of a test suite, the more likely the suite is to detect bugs in the program. There are several metrics for measuring the coverage of a test suite, such as the percentage of statements the suite executes or the percentage of program paths it exercises. In this paper, we concern ourselves with *mutation coverage*, a measure of a test suite's ability to distinguish a piece of code from small variants, or *mutants*, of that code.

Mutation testing refers to the process of evaluating the mutation coverage of a test suite. A catalog of mutations operators is applied to the program to generate a large collection of mutants, which are then executed on inputs from the test suite. If a mutant behaves differently from the original program on at least one of the tests in the suite, the suite is said to *kill* that mutant. The percentage of mutants killed is the suite's *rate of mutant killing*.

The central premise of mutation testing is the presence of a *coupling effect* between simple and complex faults in a program [4]. The effect says that a test suite that detects all simple faults in a program (as represented by a single mutation) will detect most complex faults. This claim has received experimental and analytical support [15, 14, 26, 16].

Mutation testing can be a long and costly process. The entire test suite must be run on every mutant, of which there can be on the order of hundreds to thousands. Furthermore, when a suite fails to detect a mutant, there are two possibilities: either the suite is inadequate to cover that mutant, or the mutant is equivalent to the original program. Determining if a mutant is equivalent usually involves manual inspection, although a number of automated heuristics have been proposed [19, 10, 17, 22]. Such *equivalent mutants* behave the same as the original program on every input — they cannot be killed by any test — and therefore, they do not indicate a weakness of the test suite. Equivalent mutants result from irrelevant mutations, such as negating the argument to an absolute value function or mutating dead code.

The drawbacks of achieving high coverage with testing can be mitigated through the use of *automatic test data generation*. *Constraint-based* automated test data generation is a hybrid static-dynamic approach. It statically generates test inputs guaranteed to desirable property, such as high statement, branch, or mutation coverage. These inputs can then be dynamically executed and either evaluated against an oracle, or, in the case of regression testing, kept as an oracle for future executions. Many such techniques are *constraint based*, meaning they use algebraic or logical constraint solvers to generate the test inputs.

We present a constraint-based test data generation technique for constructing a suite of test inputs that will, by construction, kill every non-equivalent mutant that is killable by some input within the user defined bounds on heap size and loop unrolling. This is facilitated by the use of a SAT-based constraint solver. Our test input generator encodes the procedure in a logical formula that allows a single mutation of that procedure to be enabled. Which mutation is chosen depends on the value of an unconstrained variable in the formula, and additional constraints encode the notion of killing a mutant. Our tool invokes a constraint solver to solve the formula for test inputs that are guaranteed to kill all killable mutants drawn from some catalog. Our constraint solver is based on the Alloy Analyzer, which encodes our formula in SAT and invokes a SAT-solver to find its solutions [11].

1.1 Comparison to Specification Analysis

Since our approach uses a constraint solver to search for tests that kill mutants, one might decide to instead search directly for an input that causes the procedure to violate its specification. Although doing so would only cost as much as a single iteration of our approach, there are two reasons why it may be undesirable in practice.

First, often no specification is available, either because no one is capable of or willing to write it or because it is not expressible in the given constraint language. Writing a correct logical specification is a difficult task that often requires skilled abstract thinking. In practice, the average programmer may find it easier to manually inspect a series of test executions for correctness than to write a correct, general specification for all possible executions. Neither our approach nor mutation testing require existing specifications.

Second, even in the presence of a specification, the cost of statically checking the code against the specification may be far greater than running the test suite. If so, it may be profitable to invest substantial time up front to generate a regression test suite that kills all mutants and that can be quickly run in the future.

2. APPROACH

We now discuss how to construct a relational first order logic formula whose solution is a pair – a mutation (drawn from some catalog) and a test input which kills that mutant. A mutation m kills an input s if s causes the original program p and the mutated program p^m to produce different output: if $p^m(s) \neq p(s)$. On successive iterations, that formula is amended to exclude any mutation which is killed by a previously generated input. The analysis we use requires a bounded state space, which corresponds to a bound on the size of the heap and the number of loop unrollings. If, on some iteration, the formula is unsatisfiable, then the previously generated inputs kill all mutants killable by inputs within the given bound. The analysis is complete. Every test added is guaranteed to kill at least one additional mutant.

2.1 Logical Constraints

A procedure p can be modeled as a function from pre- to post-states, where $p(s) = s'$ is true if and only if p relates input state s to output state s'. Applying mutation m_1 to procedure p yields the mutant procedure p^{m_1}. An Alloy formula can be constructed that is true if and only if there exists a mutation m_1 and a pre-state s_1 such that p and p^{m_1} behave differently on s_1 (s_1 kills p^{m_1}):

$$\exists m_1, s_1 \ . \ p(s_1) \neq p^{m_1}(s_1)$$

The solution witnesses the mutation m_1 and the pre-state s_1 that kills p^{m_1}. The input s_1 is added to the input suite, and the formula is amended to find a new test that kills a mutant not killed by the previous test:

$$\exists m_2, s_2 \ . \ p(s_2) \neq p^{m_2}(s_2) \wedge p(s_1) = p^{m_2}(s_1)$$

The pre-state s_2 found by this analysis kills a new mutant m_2 that would not have been killed by s_1. We then repeat this process. The formula that finds the nth such input is as follows:

$$\begin{aligned}\exists m_n, s_n \ . \quad & p(s_n) \neq p^{m_n}(s_n) \wedge \\ & p(s_{n-1}) = p^{m_n}(s_{n-1}) \wedge \cdots \wedge p(s_1) = p^{m_n}(s_1)\end{aligned}$$

This process continues until the formula is unsatisfiable, at which point a suite of test inputs has been assembled, each of which kills at least one unique mutant. One could instead halt the process after it generates a certain number of test cases or has run for a certain amount of time, but doing so risks dramatically reducing the value of the resulting test suite.

If one is generating a test suite to check the correctness of p, the correct output for each input must be determined separately. If one is generating a regression test suite (and assuming the original procedure p to be correct), each output $p(s_i)$ witnessed by the solution is considered to be the correct output.

2.2 Parameterizing Mutations

The procedure p^m is the original procedure p parameterized by the choice of a single mutation m. Thus, there are as many values m can assume as there are mutants. To construct the body of p^m, we instrument each statement in p with additional conditionals that test the value of m and that either execute the mutation to which m corresponds, or that execute the original statement if m does not correspond to a mutation. To illustrate, consider the following two program statements:

```
x = y + z;
b = x > 10;
```

Consider two mutation operators: 1) replacing an addition or subtraction sign with the other (the “ASR” mutation

Operator	Description
ASR	addition subtraction replacement
COR	comparison operator replacement
IAR	invocation argument reordering
NEG	boolean negation
OBO	off-by-one errors (± 1)
VIR	variable identifier replacement

Table 1: Mutation Operators

operation); and 2) swapping one comparison operator with another (COR). If the prior two statements appeared in the original procedure p, applying ASR to the first statement and COR to the second would yield the following statements in p^m:

```
if (m == m1)
    x = y - z;
else
    x = y + z;

if (m == m2)
    b = x >= 10;
else if (m == m3)
    b = x < 10;
else if (m == m4)
    b = x <= 10;
else if (m == m5)
    b = x == 10;
else if (m == m6)
    b = x != 10;
else
    b = x > 10;
```

Given a particular value of m, p^m simulates p varied by a single mutation. Table 1 gives the full list of mutation operators our implementation considers.

ASR swaps $+$ and $-$ operators in arithmetic expressions. COR changes one inequality comparison ($>$, $\geq$, $<$, $\leq$) into another inequality or into an equality comparison ($=$, $\neq$). COR can also toggle $=$ and $\neq$ operators. IAR reorders arguments in a procedure call if their types are compatible. NEG negates a boolean expression. OBO adds or subtracts 1 from an arithmetic expression. VIR replaces one variable identifier with another in-scope variable of the same time.

Offutt, Rothermel, Untch, and Zapf propose 22 operators for C programs, which is often used as a standard [18]. 6 correspond to our 6 (although not one-to-one). 6 more involve array references and thus would not affect the programs we have examined so far. 4 are specific to language features (e.g. GOTO). Our technique could be extended to handle the remaining 6, which include mutantions such as statement deletion and constant-for-constant replacement.

2.3 Constraint Solving

Our implementation encodes the original and mutation- parameterized procedures and the formulas presented in Section 2.1 into a relational first order logic based on Alloy [11]. It uses the encoding formalized by Dennis et al. [6], and solves it using a new tool based on the Alloy Analyzer. A solution to the formula indicates a mutant that is not killed by any previous test input, and a test input that kills that mutant. If the formula is not satisfiable, then no additional test inputs are necessary to kill all killable mutants.

The Alloy Analyzer does not symbolically prove the existence of a solution. Instead, it exhaustively searches the entire state space of scenarios within finite, user-defined bounds. It is able to analyze millions of scenarios in a matter of seconds. Consequently, failure to produce a solution does not constitute proof that one does not exist, but every solution reported is guaranteed to be correct. Failure to produce a solution does, however, guarantee that there is no solution within the chosen bounds.

Thus, the resulting test suite depends on bounds we place on the analysis. Those bounds are limitations on the number of instances of each type, in effect a limitation on the size of the heap. If a mutant can only be killed by inputs that exceed those bounds, then it will not be killed by the resulting suite.

2.4 Jimple Representation and Limitations

The programming language accepted by our tool is a subset of Java. The subset we use does not yet include arrays, iterators, constructors, exceptions, or threads. To render Java bytecode amenable to analysis we first convert it to the Jimple intermediate representation using the Soot compiler optimization framework [23, 25]. Unlike Java source code, Jimple is a typed 3-address representation in which each subexpression is assigned to a unique local variable. Other differences in Jimple include the presence of gotos in place of loop constructs, the representation of booleans as integers, the desugaring of short-circuiting boolean operators into nested if-branches, and the expanded scope of each local variable to the entirety of the method body.

As a consequence of the Jimple representation, a single mutation to the Jimple does not always correspond to the same mutation in the original source code. For example, testing whether a boolean expression is true is represented in Jimple as testing whether an integer expression is not equal to zero. Thus, using the COR mutation on Jimple to invert this equality test actually subsumes the NEG mutation at the source code level. There are also some simple source mutations that cannot be readily accomplished via Jimple, such as replacing one short-circuiting boolean operator with another. Also, because Jimple assigns each subexpression to a new local variable, applying VIR to Jimple could be equivalent to a source mutation that replaces one subexpression with another. Due to the absence of scopes on local variables, VIR can occasionally cause mutations that would have prevented the original source code for compiling successfully.

2.5 Preprocessing

Once a Java method is converted to Jimple, some additional pre-processing is required before it can be encoded as a relational model. First loops are unrolled for a finite number of iterations. At the end of the unwinding, an “assume” statement that contains the negation of the loop condition is inserted. These assume statements are recognized by our

```
1  int abs (int x) {
2    if(x >= 0)
3      return x;
4    else
5      return -x;
6  }
```

Figure 1: Absolute Value Method

encoding and constrained to be true in the resulting formula. The effect is to exclude all executions of the procedure if it would exceed the fixed number of loop iterations. Thus, if a mutant could only be killed by a test that exceeded that limit, it would not be killed by this analysis.

Lastly, dynamic dispatch is resolved into a series of tests on the type of the receiver argument, each test of which is followed by an invocation of the concrete method provided by that type. A simple class hierarchy analysis is currently used to determine the potential target methods of an invocation, though more sophisticated analyses could be easily plugged in.

2.6 Small Example

Consider the absolute value algorithm shown in Figure 1. Our mutation-parameterization instruments this code with 10 possible mutations. COR spawns five mutants on line 2: the $\geq$ can be replaced with either $>$, $<$, $\leq$, $=$, or $\neq$. The return statement on line 3 spawns two OBO mutants: `return x+1` or `return x-1`. Due to the Jimple intermediate representation of the procedure, the single Java statement `return -x` (line 5) is represented as two statements: `int x' = -x;` and then `return x';`. As a result, there are two local variables in scope (`x` and `x'`), and OBO and VIR create three mutant return values for line 5: `return x'+1`, `return x'-1`, and `return x`.

The test input generator produces 2 test inputs to kill the 10 possible mutants: 4 and -2. On this tiny example, our implementation runs nearly instantaneously. The tool first finds that the test input "$x = 4$" kills the mutant in which $\geq$ is replaced by $<$. It then amends the constraint problem to disallow any mutant killed by that input. Solving the amended formula produces the input "$x = -2$" to kill the mutant where $\geq$ is replaced by $\neq$. Amending the constraint problem a second time produces an unsatisfiable formula, indicating that no additional test inputs with scope would kill mutants which haven't already been killed.

3. RESULTS

3.1 Evaluation On Benchmarks

We evaluate our technique on several small examples, recording the execution time and number of test inputs generates. The primary factor determining the technique's run time appears to be that number of paths in the program, not the number of lines or eventual size of the test input. Since each mutation adds a conditional to the program, the number of mutations is a good estimate of the number of paths and thus of the time it takes our technique to operate.

Absolute Value

```
boolean contains(int value) {
  Bucket current = this.getHead();
  while (current != null) {
    if (value == current.getValue()) {
      return true;
    }
    current = current.getNext();
  }
  return false;
}
```

Figure 2: List Containment

In Figure 1 of Section 2.6, we saw a java procedure for computing absolute value.

6 lines of code
8 single mutants
integers range -10 to $+10$
2 inputs generated
1.5 seconds to compute those inputs

List Containment

Figure 2 gives Java code for determining whether or not an integer list contains a given value.

10 lines of code
8 single mutants
3 loop unrollings, 3 element lists, integers range -1 to $+2$
3 inputs generated
1.5 seconds to compute those inputs

Tree Node Insertion

The Appendix gives Java code for inserting a node into a binary search tree of integers.

21 lines of code
40 single mutants
3 loop unrollings, 3 node trees, integers range -2 to $+2$
5 inputs generated
30 minutes to compute those inputs

Tree Node Removal

The Appendix gives Java code for removing a node from a binary search tree of integers.

50 lines of code
242 single mutants
1 loop unrolling, 3 node trees, integers range -2 to $+2$
timed out after 10 hours, at which point it had generated 8 inputs

3.2 Compound Mutant Coverage

The *coupling effect* is the hypothesis that a suite killing most simple mutants will kill most complex ones. We evaluate this hypothesis by evaluating the input suites we generate, which kill all single mutants, against compound mutants, programs in which several of our mutation operators have been applied. The 2 inputs generated for absolute value killed all 97 possible compound mutants. The 3 inputs generated for list containment killed all 136 possible compound mutants. These are encouraging, but far from definitive, results.

3.3 Locally Minimal Test Suites

While our technique guarantees that no two inputs will kill the same set of mutants, it makes no guarantee that the suite will be globally minimal, or even locally minimal. It is guaranteed only that no two inputs in the suite kill exactly the same set of mutants. However, in the limited cases we have examined so far, the suites generated are locally minimal.

For the list containment procedure, our technique generated 3 inputs which killed all 8 possible single mutants. There are 136 compound mutants – 26 double, 44 triple, 41 quadruple, 20 quintuple, and 4 sextuple. Only 11 of those compound mutants are killed by only one of the three test inputs. 8 of those cases only killed by input #2, 2 were only killed by input #3, and 1 was only killed by input #3. Thus, of the 3 possible reduced suites and of the 136 possible compound mutants, the reduced suite will fail to catch a mutant which could have been caught by the full suite $\frac{11}{136(3)} = 2.7\%$ of the time.

If one somehow knew to leave out just input #1, the least important input, the reduced suite would only fall short of the full suite $\frac{1}{136(1)} = 0.7\%$ of the time. However, if one eliminated just input #2, the most important input, the reduced suite would fall short of the full suite $\frac{8}{136(1)} = 5.9\%$ of the time.

4. RELATED WORK

Mutation Testing and Coverage

Our approach is unique in that it is guaranteed to generate a suite of test inputs with 100% mutation coverage (of mutants killable by small inputs); other techniques rarely go above 90% coverage and are often much lower. With other techniques, generating a suite with a higher rate of mutation coverage means generating a larger suite, and achieving 100% coverage would require effectively infinite cost (detecting equivalent mutants is undecidable, and many algorithms involve a random component). With our approach, 100% of killable mutants are always killed at finite (but high) cost. Future work includes more thoroughly evaluating the compound mutant killing capabilities of suites with 100% mutation coverage and those with high but imperfect coverage.

Mutation coverage is just an estimate of error coverage, and a suite that kills many mutants could be poor at detecting real errors. It is still an open question as to just how effective mutation coverage is as a metric for evaluating test suite. While practicioners often reflect positively on mutation testing, there have been only few studies to directly support its value, and they cannot be considered definitive.

The DeMillo [2] published the first analytical and empirical evaluation of mutation coverage as a metric, which was later expanded upon by Offutt [21]. Wah [26] and Offutt [16] have worked on directly evaluating the coupling effect, the hypothesis that a suite with detects all small errors (single mutants) will be effective at detecting most complex bugs. We have added to that work by evaluating the ability of suites that kill all single mutants at killing compound errors (multiple-mutants).

For background on mutation testing, beyond what is necessary for this paper, we recommend Offutt and Untch's 2000 survey paper [22], which provides a clear and thorough discussion of the the practical obstacles and recent innovations of mutation testing.

Constraint-Based Test Data Generation

DeMillo and Offutt coined the term "constraint-based automatic test data generation" (CBT) [5] Most such techniques involve constructing a set of algebraic constraints to encode a both well-formedness and a goal (such as covering a particular statement or path) as an objective function. A minimal solution to those constraints corresponds to finding a well-formed input that achieves the goal. A much smaller set of techniques instead use logical constraints. We discuss both types below.

Test Data Generation with Algebraic Constraints

DeMillo and Offutt [5, 4, 19, 17] propose a particular CBT technique that, given a mutant, uses an algebraic constraint solver to attempt to generate a test case to kill that mutant. It tries to generate test input with the following three properties: *Reachability condition* – the program reaches the mutated statement; *Necessary condition* – the mutated statement causes the program to enter a different (presumably erroneous) state; *Sufficient condition* – that difference propagates to the output of the program. They solve algebraic constraints on the test input to meet the reachability and necessary conditions. In contrast, we directly solve the sufficient condition, implicitly satisfying both the reachability and necessary conditions. Unlike our approach, their technique attempts to kill a mutant even if that mutant could be killed by a prior test case.

In follow-up work, Offutt and Pan [19] present a technique for detecting that two mutants are equivalent. Like DeMillo and Offutt's automatic test case generation, it is a heuristic-based method for solving algebraic constraints that is unsound. Because our technique searches for a test case that causes the mutant to behave differently than the original, each mutant found is non-equivalent by construction.

Ferguson and Korel propose *chaining*, a method to generate tests by observing the execution of the program under an existing test [7]. They bias the algorithm towards generating tests that provide coverage of particular lines of code – either lines of particular interest or lines that are not covered by the existing tests. To that end, they use data dependence analysis to guide the search process by identifying statements that affect the execution of the target statement. They evaluate it in terms of the likelihood of finding well formed inputs that exercise the target line.

Tracey, Clark, and Mander introduce a technique which solves algebraic constraints to automatically generate well formed test data [24]. They theorize that, while solving general constraints is intractable, constraints resulting from real software are a much easier subset. They solve the constraints with the help of *simulated annealing*, a probabilistic algorithm for global optimization of some objective function inspired by metallurgy models of cooling. They construct an objective function which represents how close a given solution is to satisfying the algebraic constraints (e.g. $x = 49$ is closer to satisfying $X > 50$ than is $x = 2$). Randomization in the algorithm causes it to (usually) produce different results on different iterations. They evaluate their work in terms of the time taken by their implementation to generate 50 well formed test inputs for small examples. Their approach is focused on generating well formed inputs given a set of algebraic constrains, not inputs with any particular feature (e.g. mutation coverage).

Gupta, Mathur, and Soffa show how *iterative relaxation* an be used to generate test inputs that cover a specified path, as a step in a building a test suite with path coverage. An input is iteratively refined until all branch predicates along a given path evaluate to the desired outcome. In each iteration, the current input is executed and used to derive a set of linear constraints. Those constraints are solved (using Gausian elimination) to obtain the input for the next iteration. If the branch conditions on the given path are linear, the the algorithm will terminate with either an input which takes that path, or a guarantee that the path is infeasible. The complexity of the problem grows with the number and complexity of the branch predicates, not the length of the trace, so the authors are optimistic about the practical scalability of the algorithm. [8]

Michael and McGraw developed the the GADGET system which uses a form of dynamic test data generation [13]. It treats parts of the program as functions which can be evaluated by executing the program, and whose value is minimal for inputs that satisfy some desirable property (in their paper, the use condition coverage). Because of that correspondence between test data generation and function minimization, they can instead solve the latter, better-understood problem using techniques such as simulated annealing, genetic search, and gradient descent. A randomized component encourages the solutions to be different on different runs. The authors evaluate the different methods of solving the algebraic constraints generated by GADGET in terms of condition coverage of a 2000 line C program, with respect to the number of tests generated. The curves flatten out at about 10,000 test inputs, with the leaders being simulated annealing (91%) and gradient descent (83%). The time taken to generate these tests is not reported.

Logical Constraint Solving

Marinov and Khurshid introduced *TestEra* [12] and Boyapati, Khurshid, and Marinov introduced *Korat* [1] – tools that use logical constraint solving to generate inputs to a program based on a well-formedness predicate on the input. Symmetry breaking constraints added to the constraint problem prevent the generation of isomorphic input. Inputs are not filtered according to any particular metric. In contrast, our approach examines the program body to select a much smaller subset of the inputs that still achieves mutation coverage. Korat, like our tool, uses Alloy technology to solve logical constraints.

5. REFERENCES

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. Submitted for publication, February 2002.

[2] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233, New York, NY, USA, 1980. ACM Press.

[3] W. H. Deason, D. B. Brown, K. H. Chang, and J. H. Cross II. A rule-based software test data generator. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):108–117, 1991.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practical programmer. 11(4):34–41, April 1978.

[5] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, September 1991.

[6] G. Dennis, F. Chang, and D. Jackson. Checking refactorings with SAT. Submitted for publication, September 2005.

[7] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[8] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 231–244, New York, NY, USA, 1998. ACM Press.

[9] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[10] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification & Reliability*, 9(4):233–262, 1999.

[11] D. Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering (FSE)*, November 2000.

[12] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of Java programs. In *Proc. 16th International Conference on Automated Software Engineering (ASE)*, November 2001.

[13] C. C. Michael and G. McGraw. Automated software test data generation for complex programs. In *Automated Software Engineering*, pages 136–146, 1998.

[14] L. J. Morell. A theory of fault-based testing. *IEEE Trans. Softw. Eng.*, 16(8):844–857, 1990.

[15] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[16] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[17] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification & Reliability*, 4(3):131–154, 1994.

[18] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[19] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test., Verif. Reliab.*, 7(3):165–192, September 1997.

[20] A. J. Offutt, J. Voas, and J. Payne. Mutation operators for Ada, 1996.

[21] J. Offutt, A. Lee, G. Rothermel, R. Untch, and Author:. An experimental determination of sufficient mutation operators. Technical report, Fairfax, VA, USA, 1994.

[22] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, October 2000.

[23] V. S. P. L. E. G. Raja Vallée-Rai, Laurie Hendren and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[24] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 73–81, New York, NY, USA, 1998. ACM Press.

[25] R. Vallee-Rai. The Jimple framework. 1998.

[26] K. S. H. T. Wah. An analysis of the coupling effect I: single test data. *Sci. Comput. Program.*, 48(2-3):119–161, 2003.

Appendix

Java code for a binary search tree of integers with insert and remove methods.

```
package test;

class Node {
    int value;
    Node left, right;

}

class Tree {
    Node root;

    void insert(Node n) {
        Node x = this.root;
        Node parent = null;

        while (x != null) {
            parent = x;
            if (n.value < x.value) {
                x = x.left;
            } else {
                x = x.right;
            }
        }

        if (parent == null) {
            this.root = n;
        } else {
            if (n.value < parent.value) {
                parent.left = n;
            } else {
                parent.right = n;
            }
        }
    }

    boolean remove(int info) {
        Node parent = null;
        Node current = root;

        while (current != null) {
            if (info == current.value)
                break;
            if (info < current.value) {
                parent = current;
                current = current.left;
            } else /* (info > current.value) */ {
                parent = current;
                current = current.right;
            }
        }

        if (current == null) return false;
        Node change = removeNode(current);

        if (parent == null) {
            root = change;
        } else if (parent.left == current) {
            parent.left = change;
        } else {
            parent.right = change;
        }
        return true;
    }

    static Node removeNode(Node current) {
        Node left = current.left, right = current.right;

        if (left == null)
            return right;
        if (right == null)
            return left;
        if (left.right == null) {
            current.value = left.value;
            current.left = left.left;
            return current;
        }

        Node temp = left;
        while (temp.right.right != null) {
            temp = temp.right;
        }

        current.value = temp.right.value;
        temp.right = temp.right.left;
        return current;
    }
}
```