Predictive Testing: Amplifying the Effectiveness of Software Testing

Pallavi Joshi
Koushik Sen
Mark Shlimovich

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-35
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-35.html

March 20, 2007

Predictive Testing: Amplifying the Effectiveness of Software Testing

Pallavi Joshi
EECS Department
University of California,
Berkeley, USA
pallavi@eecs.berkeley.edu

Koushik Sen
EECS Department
University of California,
Berkeley, USA
ksen@cs.berkeley.edu

Mark Shlimovich
EECS Department
University of California,
Berkeley, USA
mark@shlim.net

ABSTRACT

Testing with manually generated test cases often results in poor coverage and fails to discover many corner case bugs and security vulnerabilities. Automated test generation techniques based on static or symbolic analysis usually do not scale beyond small program units. We propose predictive testing, a new method for amplifying the effectiveness of existing test cases using symbolic analysis. We assume that a software system has an associated test suite consisting of a set of test inputs and a set of program invariants, in the form of a set of assert statements that the software must satisfy when executed on those inputs. Predictive testing uses a combination of concrete and symbolic execution, similar to concolic execution, on the provided test inputs to discover if any of the assertions encountered along a test execution path could be violated for some closely related inputs. We extend predictive testing to catch bugs related to memory-safety violations, integer overflows, and string-related vulnerabilities.

Furthermore, we propose a novel technique that leverages the results of unit testing to hoist assertions located deep inside the body of a unit function to the beginning of the unit function. This enables predictive testing to encounter assertions more often in test executions and thereby significantly amplifies the effectiveness of testing. We have implemented predictive testing in a tool called PRETEX and our initial experiments on some open-source programs show that predictive testing can effectively discover bugs that are missed by normal testing. PRETEX uses symbolic analysis and automated theorem proving techniques internally, but all of this complexity remains hidden from the user behind a testing usage model. For this reason, we expect that PRETEX will be easy to integrate into existing software engineering processes and will be usable even by unsophisticated developers.

1. INTRODUCTION

Software testing usually accounts for 50% to 80% of the software development cost. Generation of test inputs is an expensive key component of software testing. Manual techniques are predominantly used in software industry to generate test inputs. Unfortunately, testing using manually generated test inputs often results in poor coverage and fails to find many corner case bugs and security vulnerabilities resulting from buffer overflows, integer overflows, etc.

In order to improve test coverage, several techniques have been proposed to automatically generate values for the test inputs. *Random testing* generates test inputs by randomly picking values from some potential input domain [3, 10, 7, 21]. The problem with such random testing is two fold: first, many sets of values may lead to the same observable behavior and are thus *redundant*, and second, the probability of selecting particular inputs that cause buggy behavior may be astronomically small [20]. Test input generation based on *symbolic execution* [14, 26, 1, 13, 25, 2, 27, 8, 24] addresses the problem of poor test coverage. In symbolic execution, a program is executed using symbolic variables in place of concrete values for inputs. Each conditional expression in the program represents a constraint that determines an *execution path*. Test inputs are generated for each path by solving the symbolic constraints along the path using an automated theorem prover. Unfortunately, for large or complex units, it is computationally intractable to precisely maintain and solve the constraints required for test generation.

To address the limitations of pure symbolic execution, concolic testing [11, 22, 4, 5, 17] has been proposed recently. Concolic testing, which is based on dynamic methods for test input generation [15, 23], iteratively generates test inputs by combining concrete and symbolic execution. Concolic testing shows how static and dynamic program analysis can be combined with rigorous model-checking techniques to automatically and systematically generate test inputs. Unfortunately, concolic testing does not scale for large programs, because often the number of feasible execution paths of a program increases exponentially with the increase in the length of an execution path. We call this the *path explosion problem*. Therefore, the approach remains limited to unit testing and cannot be readily applied to test an entire system.

We propose *predictive testing*, a more pragmatic approach to testing. Rather than trying to generate test inputs automatically, which is typically infeasible for large programs, we assume that a large program already comes with a test suite, where a test suite consists of a set of test inputs and global invariants present in the form of a set of assertions throughout the program. Predictive testing tries to *amplify the effectiveness of testing* the program on the provided test

suite by transparently performing symbolic execution and automated theorem proving along an actual test execution path on a given test input, similar to concolic execution. The key insight is that if we analyze the trace of a successful test execution, we can often spot assertions that would fail in a "nearby" execution for slightly modified inputs. This enables us to test if a property *may* have been violated, even though it was not observed to be violated in the original test execution.

Specifically, at the beginning of a test execution, we create a symbolic state by initializing the inputs with symbolic values, in addition to the normal concrete state of the program. We then execute the program both concretely and symbolically, similar to concolic execution. At the execution of every conditional statement in the concrete execution, the symbolic execution generates symbolic constraints over the symbolic input variables. At the execution of an assert statement, unlike concolic testing, the symbolic execution invokes a theorem prover to check if the assertion is guaranteed by the conjunct of branch constraints encountered so far. If the assertion is not guaranteed, we predict a potential violation of the assertion. Observe that even if the concrete execution passes the assertion, we can predict violation of the assertion for some closely related input along the same execution path.

Apart from checking the user provided assertions, our predictive testing implementation, inserts its own assertions to check various generic properties such as memory-safety and absence of integer overflow and string vulnerabilities. For memory safety checks, we use a variant of Jones Kelly's [12] backward compatible dynamic memory safety checking technique. Although we currently predictively check the above generic properties, checks for other generic properties can easily be integrated into our framework provided that there is a technique to generate assertion statements for such checks.

A novel contribution of this work is *assertion hoisting*. Predictive testing as described above tries to verify a test execution path for all possible inputs. Only assertions that are actually executed on some test execution are analyzed to be violated. To further increase the effectiveness of predictive testing, we perform assertion hoisting before performing actual predictive testing, i.e., we move assertions from various parts in the body of a unit or a function to the beginning of the unit or the function so that the assertions can be encountered more often by test executions. We assume that a unit has a unit test suite.[1] We perform a combination of concrete and symbolic execution of the unit on the given test inputs and generate a precondition for the unit. The precondition is added to the beginning of the unit as an assertion, so that it can be encountered by more test executions during predictive testing of the entire program. Figure 1 gives a diagrammatic comparison between traditional testing, predictive testing, and predictive testing with assertion hoisting.

We have implemented the predictive testing technique for C programs in a prototype tool called PRETEX. Our initial experiments on a set of medium-sized programs with existing test suites show that PRETEX can predict bugs that are missed by normal testing. We found that an automated test generation tool, such as CUTE, fails to generate meaningful test inputs for these programs in reasonable amount of time, such as one day.

[1]Such a test suite can also be generated using existing automated test generation techniques.

```
int dbl(int z) {
   return 2*z;
}

void main() {
1:   int x, y, u, v;
2:   x = input();
3:   y = input();
4:   u = dbl(x);
5:   v = dbl(y)+1;
6:   if (u > v) {
7:      u = u - 1;
     }
8:   assert(u != v);
}
```

Figure 2: A simple example to illustrate PRETEX. We assume that we have a test-suite containing the test inputs {x=3, y=9} and {x=-6, y=-100}. The assert at line 8 passes on these test inputs. However, predictive testing on these test inputs discovers potential violation of the assertion.

PRETEX uses complex static analysis and automated theorem proving techniques internally, but all of this complexity is hidden from the user by a testing usage model. For this reason, we expect that PRETEX will be easy to integrate into existing software engineering processes and will be usable even by unsophisticated developers.

Our work on predictive testing is related to the work of Larson and Austin [16]. Larson and Austin maintain range constraints over array sizes during a concrete execution of programs and predicts buffer overflow errors. The key differences between their approach and PRETEX are the following. We look for any error that can be translated into assertions, including buffer overflows. We use full fledged symbolic execution and automated theorem proving, rather than simple range constraints. We use assertion hoisting that enables us to verify assertions, even if they are not encountered along an execution path. For example, Larson and Austin's technique will be not able to discover bugs in all of the three examples in Section 2.

2. OVERVIEW

In this section, we will give a gentle introduction to PRETEX. We will start with a simple example and illustrate how PRETEX uses a combination of concrete and symbolic execution, similar to concolic execution, on existing test inputs to predict potential bugs rather than generating new test inputs. Then we will illustrate how PRETEX can predict memory safety errors in programs by looking at their successful executions. Finally, we will show how we utilize unit testing results to significantly increase the effectiveness of PRETEX by using assertion hoisting.

2.1 Simple Predictive Testing

Consider the code in Figure 2. The code defines a `main` function, which takes two inputs `x` (line 2) and `y` (line 3). We use the statement `m = input();` to indicate that the variable `m` is assigned an external input. The function `main` assigns `dbl(x)` to `u` (line 4) making it an even number and assigns `dbl(y)+1` to `v` (line 5) making it an odd number. If

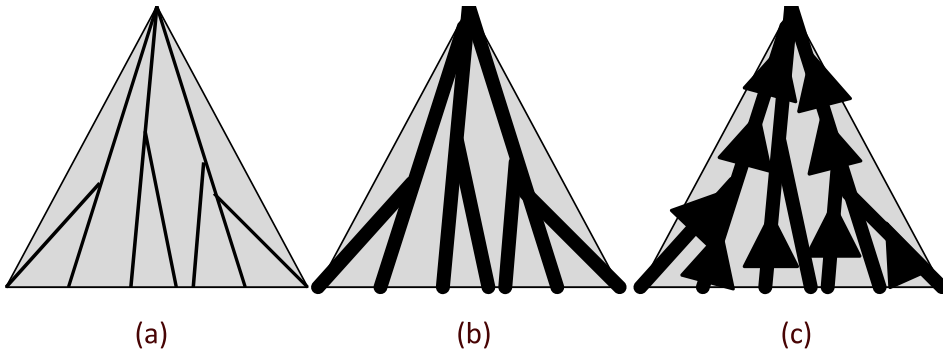

Figure 1: Comparison between (a) Traditional Testing (b) Simple Predictive Testing (c) Predictive Testing with Assertion Hoisting. In the above figures, the triangular shaded regions represent the space of all execution paths of a program. The black lines or regions represent the space covered by each testing technique. Traditional testing (a) only covers a few black lines or execution paths for a given test suite. Predictive testing (b) on the same test suite expands each such line or execution path by checking all possible inputs along that path. Predictive testing with assertion hoisting (c) further increases the coverage by checking "nearby" paths.

`u > v` holds at line 6, then `u` is decremented by 1. Finally, the program asserts that `u != v`, which can be seen as an invariant that the program must satisfy at line 8.

Although the program is an artificial one, it illustrates several key components that a large program can have such as *inputs* and *assertions specifying the correctness requirements of the program.* We assume that the program in Figure 2 also comes with a test-suite containing the test inputs {`x=3, y=9`} and {`x=-6, y =-100`}. For large programs, such test-suites are usually manually generated; testing using such test-suites may not expose all bugs. Test inputs that can expose all bugs can be generated automatically using sophisticated program analysis techniques; however, the scalability of such techniques remains limited to small programs or units only. Instead of trying to generate test inputs automatically, PRETEX tries to improve the effectiveness of an existing test-suite by exploiting program analysis techniques previously used for the purpose of test input generation. We next illustrate this.

If we execute `main` on the given test inputs, the assertion will succeed on both executions. However, the assertion can be violated for some inputs such as {`x=2, y=1`}. PRETEX will discover that the assertion can be violated on a closely related input. It does this by augmenting the concrete execution of the program on the given test input {`x=-6, y=-100`} with symbolic execution as follows:

- PRETEX creates a symbolic state in addition to the concrete state or the normal state of the program. PRETEX initializes the inputs with symbolic values in the symbolic state. For example, PRETEX assigns symbolic values x and y to `x` and `y`, respectively, in the symbolic state. In the concrete state, PRETEX assigns `-6` and `-100` to `x` and `y`, respectively. The symbolic state encodes all possible values that variables derived from program input can take on at each point along the current execution path.

- For all assignment operations of the form `lhs = e` along the current execution path, PRETEX evaluates and assigns the expression `e` symbolically in the symbolic state and concretely in the concrete state. For example, after the execution of the statements at line 4 and 5, the symbolic state gets updated to $\{\texttt{u} \mapsto 2*x$, $\texttt{v} \mapsto (2*y+1)\}$, whereas the concrete state becomes $\{\texttt{u} \mapsto -12$, $\texttt{v} \mapsto -199\}$. Similarly, after the execution of the statement at line 7, the symbolic state gets updated to $\{\texttt{u} \mapsto (2*x-1)$, $\texttt{v} \mapsto (2*y+1)\}$.

- For every conditional encountered during the execution, PRETEX evaluates the predicate in the conditional both in the symbolic state and concrete state. The result of the evaluation in the concrete state is used by the concrete execution to determine the next program counter (i.e., to determine the execution path.) The result of the evaluation in the symbolic state results in a constraint over the symbolic input variables. This constraint is appended to a propositional formula, called the *path constraint*, as a conjunct. The path constraint is initialized to *true* at the beginning of the execution. The path constraint expresses the set of inputs for which the program would take the same execution path as the concrete test input. For example, the conditional at line 6, results in the path constraint $(2*x) > (2*y+1)$.

- For every assertion encountered along the concrete execution, PRETEX checks if the assertion is satisfied for all input values that would satisfy the symbolic path constraint. Hence, if it PREFIX passes an assertion, it is guaranteed that every input that results in the same execution path as the test input will also pass this assertion. To achieve this, PRETEX evaluates the predicate inside the `assert` in the symbolic state and uses an automated theorem prover to check that the current path constraint implies the result of the evaluation; if it does not, PRETEX reports a bug. For example, at line 8, PRETEX evaluates the predicate inside the `assert` to $(2*x-1)! = (2*y+1)$ and checks if the current path constraint $(2*x) > (2*y+1)$ implies $(2*x-1)! = (2*y+1)$. It does not, and the theorem prover returns $\{x = 2, y = 1\}$ as a counter-example. Therefore, PRETEX reports a bug. Observe that the concrete execution (the original test case) satisfies the

```
main(){
1:   unsigned int size, index, *array;

2:   size = getchar();
3:   if (size <= 0 || size > 10) exit();
4:   array = (unsigned int *) calloc(size, sizeof(unsigned int));
5:   /* initialize array */
6:   index = getchar();
7:   if (index < 0 || index > 9) exit();
8:   assert(index>=0 && index < size);
9:   y = array[index];
}
```

Figure 3: Predicting memory safety error at line 9 using the test input {size=5, index=3}. The assert at line 8 is inserted automatically by PRETEX. The assert passes on the given test input. However, PRETEX discovers a potential violation of the assertion.

check in the assertion.

PRETEX exposes the bug from a successful test execution. In the preceding example, even though the actual execution does not violate the assertion (i.e., does not expose the bug), PRETEX discovers the bug because it tries to verify the current execution path for all potential input values resulting in the same path. Thus using a single test input, predictive testing amplifies the effectiveness of testing by considering all closely related inputs that were not considered during normal testing.

2.2 Predictive Testing for Memory Safety Errors

Memory safety is an important property because it is an absolute prerequisite for soundness of other analyses, and it is also often the root cause of many security vulnerabilities. We next illustrate how predictive testing can be used in conjunction with existing memory-safety error detection tools to predict memory-safety errors from error free test executions. Consider, the `main` function in Figure 3 (obtained from [16]), which gets two inputs using the function `getchar()` and assigns them to the variables `size` and `index`, respectively. If the user input `size` is in the range $[1, 10]$, then an array of length `size` is created (line 4). Then the array is initialized; this code is omitted to keep the program simple. The `index` element of the array is read at line 9. The assert statement at line 8 was originally not present in the code and has been inserted by PRETEX to detect memory-safety bugs. Several techniques can be used to insert such assertions; PRETEX uses a technique similar to one proposed by Jones and Kelly [12]. Further details about this technique is discussed in Section 3.4.

Suppose, we have a test-suite containing a test input {`size=5, index=3`}. An execution of the `main` on this test input will not violate the assertion; therefore, no memory-safety error will be detected. However, if we perform predictive testing on this successful test execution by augmenting the concrete execution with symbolic execution, we can immediately show that there is a memory-safety error in the code. To do so, PRETEX creates a symbolic state where it assigns the symbolic values *size* and *index* to the variables `size` and `index`, respectively. PRETEX also creates the concrete state with `size=5` and `index=3`.

PRETEX next executes the `main` function both concretely and symbolically. After the execution of the statement at line 7, the path constraint becomes ($size > 0 \wedge size \leq 10 \wedge index \geq 0 \wedge index \leq 9$). In the concrete execution, the assertion at line 8 is not violated as `size=5` and `index=3`. In the augmented symbolic execution, the predicate inside the assertion gets symbolically evaluated to ($index \geq 0 \wedge index < size$). PRETEX then checks if this symbolic constraint is implied by the current path constraint ($size > 0 \wedge size \leq 10 \wedge index \geq 0 \wedge index \leq 9$). This is false and the counter-example `size=5` and `index=6` is generated by the theorem prover. As a result, PRETEX reports a bug and provides the counter example to the tester. PRETEX predicts a memory safety error, whereas a simple memory safety checker passes the assertion when run with the same test input. Note that Larson and Austin's method, which employs a range based technique for catching memory safety errors, will not be able to catch this bug.

Any property that can be enforced by using assertions inside the program can be checked predictively by PRETEX. The memory-safety property demonstrated is just one example. In subsequent sections, we show how we can translate bugs related to integer-overflow or underflow, division-by-zero, and assignment of values of a type to a sub-type (such as integer to unsigned char) into assertions. Once the assertions are in place, PRETEX can potentially catch such bugs from successful test executions.

2.3 Exploiting Unit Testing Results in Predictive Testing

Predictive testing as illustrated above tries to verify assertions along a test execution path for all possible inputs that would result in the same path. Only assertions that are actually executed on some test execution are analyzed for possible violations. For example, consider the function `main` in Figure 4, which gets two inputs in the variables `i` and `x`, respectively. The assert statement at line 3 is inserted automatically by a memory-safety error detection tool. The assert statement at line 1 is originally not present in the code. We derive this assertion using the results of unit testing the function `set` and insert it at line 1. This technique of deriving and inserting assertions at the beginning of a unit function is called *assertion hoisting*. Suppose, we have a test-suite containing two test inputs {`i=2,x=-1`} and {`i=9,x=8`} for the function `main`. Unfortunately, the execution of `main` on these test inputs will not execute the path containing the assert statement at line 3. As such PRETEX will not be able to verify the path and it will not discover the bug.

Assume that the writer of the `set` function has unit tested the function using the test inputs {`i=1, x=0`} and {`i=3, x=100`}. Both of these test inputs result in no error in the function `set`. The writer of the function `set` knows the implicit precondition under which the function is going to behave correctly and she/he picked the test inputs so that they satisfy the precondition. The goal of *assertion hoisting* is to derive this implicit precondition by looking at the unit test executions. We next describe how PRETEX derives this precondition.

PRETEX executes the `set` function both concretely and symbolically as described in Section 2.1 on both test inputs. PRETEX also creates a symbolic formula, called precondition, which is initialized to true. If during symbolic execution, PRETEX encounters an `assert` statement, it adds

```
unsigned int array[10];

main(){
  unsigned int i, x;

  i = input();
  x = input();
  if (x > 0) {
     set(i,x);
  }
}

void set(unsigned int i,unsigned int x) {
1: assert(x != 100 || i > 10 || (i >= 0 && i < 10));
2: if (x==100 && i <= 10) {
3:     assert(i >=0 && i < 10);
       array[i] = x;
   }
}
```

Figure 4: Assertion Hoisting. The assert statement at line 1 is generated by PRETEX using the assertion hoisting technique. Execution of `main` on the given test inputs {`i=2,x=-1`} and {`i=9,x=8`} does not encounter the existing assertion at line 3. However, one of these test executions encounters the hoisted assertion at line 1. This enables PRETEX to predict bugs, whereas predictive testing without assertion hoisting fails to discover the bug.

$(\phi \Rightarrow c)$ to the precondition as a conjunct, where ϕ is the current path constraint and c is the symbolic constraint that is obtained by symbolically evaluating the predicate inside the `assert` statement. For example, the symbolic execution of the `set` function on the test input {`i=3,x=100`} generates the path constraint $(x = 100 \wedge i \leq 10)$ after the execution of the conditional at line 2. Then after the execution of the `assert` statement at line 3, PRETEX generates the precondition $(x = 100 \wedge i \leq 10) \Rightarrow (i \geq 0 \wedge i < 10)$, which simplifies to the predicate inside the `assert` statement at line 1. The symbolic execution of `set` on the input {`i=1,x=0`} results in no precondition because PRETEX encounters no `assert` statement along the concrete execution path. Therefore, based on the two test inputs for `set`, PRETEX generates the above precondition and inserts it as an assertion at the beginning of the function (line 1.)

Once PRETEX has inserted the assertion at the beginning of the `set` function, a predictive testing of the `main` function on the input {`i=9,x=8`} shows that the assertion at line 1 can be violated. Therefore, predictive testing discovers the bug.

Thus PRETEX utilizes unit testing to hoist asserts so that PRETEX can not only verify a test execution path for all possible inputs, but also verify other "nearby" paths that are not exercised during testing.

3. PREDICTIVE TESTING

In this section, we present the predictive testing technique. First, we present a modification of concolic execution that tries to verify if a test execution on a given input is correct for all possible inputs, rather than generating test inputs. Then we describe a novel technique, called *assertion hoisting*, that exploits the results of unit testing and enables us to verify not only the current execution path, but the other closely related execution paths of a program. This technique

```
Inputs: e : Expression to be evaluated symbolically
        M : Concrete state of the program
        S : Symbolic state of the program
evaluate_symbolic (e, M, S) =
  match e:
    case c :    // c is a constant
      return c
    case m:     // m is a memory location
      if m ∈ domain(S) then return S(m)
      else return M(m)
    case e' op e'':
      let f'= evaluate_symbolic(e', M, S);
      let f''= evaluate_symbolic(e'', M, S);
      if f' op f'' is not in decidable theory then
        complete = false
        return evaluate_concrete(e, M)
      else
        return f' op f''
```

Figure 5: Symbolic evaluation

significantly amplifies the effectiveness of testing by reusing the results of unit testing. Finally, we describe our extensions of predictive testing to predict memory safety errors, integer overflow, and string related errors from successful executions. These extensions help PRETEX to predict corner case security vulnerabilities that traditional testing often misses.

3.1 Simple Predictive Testing: Verifying a Test Execution Path for all Possible Inputs

We assume that we are given a large program P and a test-suite for P. A test-suite consists of a set of test inputs to the program P and a set of correctness properties of the program. The correctness properties are given in the form of assertions over the program state.

Programming Model and Concrete Semantics.

We describe our algorithm on a simple imperative language. A program P in the imperative language manipulates the memory through *statements* that are specially tailored abstractions of the machine instructions actually executed. There is a set of numbers L that denote instruction addresses, that is, statement labels. If ℓ is the address of a statement (other than `halt`), then $\ell + 1$ is guaranteed to also be an address of a statement. The initial address is ℓ_0. A statement can be a *conditional statement* of the form if (e) then goto ℓ' (where e is an expression and ℓ' is a statement label), an *assignment statement* of the form $m \leftarrow e$ (where m is a memory address), an *input statement* of the form $m \leftarrow$ `input()`, an *assert statement* of the form `assert`(e), or a `halt` statement, corresponding to normal termination. An expression in a statement has no side-effects. If e evaluates to false in an `assert`(e) statement, then the program aborts indicating an error.

The *memory* $\mathcal{M}$ is a mapping from memory addresses m to values. Given a memory $\mathcal{M}$, we use $\mathcal{M}' = \mathcal{M}[m \mapsto v]$ to denote the same map as $\mathcal{M}$, except that $\mathcal{M}'(m) = v$. The concrete semantics of the programming language is reflected in *evaluate_concrete*$(e, \mathcal{M})$, which evaluates expression e in the memory $\mathcal{M}$ and returns a value for e. Additionally, the function *statement_at*(P, pc) returns the statement with label pc in the program P.

Algorithm.

We represent a symbolic value by s_i where i is a natural number. A *symbolic expression* e can be of the form s_i, c (a constant), $e' + e''$ (a dyadic term denoting addition), $e' \leq e''$ (a term denoting comparison), $\neg e'$ (a monadic term denoting negation), $*e'$ (a monadic term denoting pointer dereference), etc. PRETEX maintains a *symbolic memory* $\mathcal{S}$ that maps memory addresses to symbolic expressions. Initially, $\mathcal{S}$ is empty. Expressions are evaluated symbolically by the function *evaluate_symbolic* described in Figure 5. When an expression falls outside the theory, as in the multiplication of two non-constant sub-expressions, *PRETEX simply falls back on the concrete value* of the expression, which is used as the result. In such cases, we set the flag *complete* to **false** to indicate that our algorithm can no longer be sound, i.e., we cannot verify the current execution path for all inputs.

execute_program in Figure 6 describes the predictive testing algorithm of PRETEX. The function takes the following as inputs. P is the function to test: it can be the `main` function, or any other function in the program. TestInp is the test input vector on which we want to test P. PRETEX first initializes the local variable pc, denoting the program counter, to the address ℓ_0 of the first statement of P. PRETEX initializes the concrete memory $\mathcal{M}$ and the symbolic memory $\mathcal{S}$ to the empty map and initializes the path condition ϕ and the precondition Φ of P to **true**.

PRETEX then executes P both concretely and symbolically. Specifically, PRETEX does the following in a loop until the program terminates normally or erroneously: PRETEX determines **s**, the next statement to be executed, and executes the statement as follows.

If the statement **s** is of the form $m \leftarrow$ `input`(), then in the concrete state $\mathcal{M}$, the value in the address m is updated by the next available input from the test input vector TestInp. The index of the next available input in TestInp is maintained in the local variable i. In the symbolic state $\mathcal{S}$, the memory m is mapped to a fresh symbolic value s_i. i is then incremented.

If the statement **s** is of the form $m \leftarrow e$, then e is evaluated both symbolically and concretely in the states $\mathcal{S}$ and $\mathcal{M}$, respectively. The results are used to update the mapping of m in both $\mathcal{S}$ and $\mathcal{M}$.

If the statement **s** is of the form if(e)then gotoℓ', then e is evaluated both symbolically and concretely in the states $\mathcal{S}$ and $\mathcal{M}$, respectively. The symbolic evaluation of e results in a symbolic constraint c over the symbolic input values. c or $\neg c$ is added to the path constraint ϕ as a conjunct depending on the outcome of the concrete evaluation of e. This path constraint expresses the set of test input vectors for which the program would take the same execution path as the concrete test input.

If the statement **s** is of the form `assert`(e), then PRETEX performs a *passive check* to predict bugs. First, PRETEX evaluates e concretely; if the result is false, then there is an actual bug in the current execution; PRETEX then aborts the current execution. Such a bug can also be discovered by normal testing on the given input TestInp. However, PRETEX can predict a bug even if the current execution does not violate the assertion. To do so, PRETEX evaluates the predicate e inside the `assert` in the symbolic state $\mathcal{S}$ and uses an automated theorem prover, represented by the function *is_satisfiable*, to check if the current path constraint implies the result of the evaluation; if not, PRETEX predicts a bug. When PRETEX predicts a bug, we know that for some other closely related test input vector the assertion fails. PRETEX also generates test input vector as evidence to the predicted bug by using a constraint solver.

```
execute_program(P, TestInp)
    pc = ℓ0; i = 0;
    M = S = [ ];
    φ = true;
    Φ = true;
    while (true)
        s = statement_at(P, pc);
        match (s)
            case (m ← input()):
                M = M[m ↦ TestInp[i]];
                S = S[m ↦ s_i];
                i = i + 1;
            pc = pc + 1;
        case (m ← e):
            v = evaluate_concrete(e, M);
            M = M[m ↦ v]; pc = pc + 1;
            S = S[m ↦ evaluate_symbolic(e, M, S)];
        case (if (e) then goto ℓ'):
            b = evaluate_concrete(e, M);
            c = evaluate_symbolic(e, M, S);
            if b then
                φ = φ ∧ c; pc = ℓ';
            else
                φ = φ ∧ ¬c; pc = pc + 1;
        case assert(e):
            b = evaluate_concrete(e, M);
            c = evaluate_symbolic(e, M, S);
            Φ = Φ ∧ (φ ⇒ c);
            if b then
                if is_satisfiable(φ ∧ ¬c) then
                    print "Bug Predicted";
                pc = pc + 1;
            else
                print "Bug Found";
                return Φ;
        case halt:
            return Φ;
```

Figure 6: PRETEX Testing Algorithm

While executing an `assert`(e) statement, PRETEX also adds the formula $(\phi \Rightarrow c)$ as a conjunct to the precondition Φ, where ϕ is the current path constraint and c is the symbolic constraint resulted by the symbolic evaluation of e. The precondition of P represents a formula that if violated by a test input vector would mean that there is a bug in P. In predictive testing, we do not use this precondition. We use this precondition for *assertion hoisting* as described in the next section.

The following result holds for the PRETEX testing algorithm.

THEOREM 1. **I***f PRETEX predicts no assertion violation along an execution path and if the flag complete is not set to false in evaluate_symbolic (i.e. PRETEX has not done any approximations), then all assertions along the execution path succeed for all inputs resulting in the same execution path.*

Differences Between Concolic Testing and PRETEX

We point out the differences and the similarities between concolic testing and PRETEX. In PRETEX, we assume that a test input vector is available and unlike concolic testing, we

```
unit_test(f, TestSuite)
    Φ = true;
    foreach (TestInp ∈ TestSuite)
        Φ = Φ∧ execute_program(f, Φ, TestInp);
    insert assert(Φ) as the first statement of f
```

Figure 7: PRETEX Assertion Hoisting Algorithm

do not try to generate a test input vector either randomly or through constraint solving. PRETEX invokes the theorem prover at every `assert` statement encountered in an execution path to check if the assertion can be violated for some closely related input along the same execution path. In concolic testing, a symbolic constraint is generated and added to the path constraint for every `assert` statement encountered along a path. Concolic testing invokes the theorem prover at the end of an execution to generate a new test input vector that forces the execution along a new unexplored path. The simultaneous concrete and symbolic execution of a program by PRETEX is similar to concolic execution in the sense that both of them maintain symbolic state and generate symbolic path constraints. However, they use the generated path constraint for different purposes: predictive testing for predicting errors and concolic testing for generating new test input vectors.

3.2 Assertion Hoisting using Unit Testing

Predictive testing as described above tries to verify a test execution path for all possible inputs. Only assertions that are actually executed on some test execution are analyzed to be violated. To further increase the effectiveness of predictive testing, we perform assertion hoisting before predictive testing, i.e., we move assertions from various parts in the body of a function to the beginning of the function so that the assertions can be encountered more often by test executions.

For assertion hoisting, we exploit the fact that unit tests are available for various units or functions of the program. Such tests are often written by the developers. In fortunate situations, such tests can be generated using automated test generation techniques as well. In PRETEX, we will not worry about how to get these unit tests, rather we will use existing unit test cases to perform assertion hoisting.

Formally, let us assume that our program P is split into several units. Let f be the entry function of one such unit. The arguments to this function are treated as inputs to the unit. Let us further assume that each unit has a corresponding test-suite. Let TestSuite be the test-suite for the function f. For each pair $(f, \mathsf{TestSuite})$, we run the algorithm in Figure 7. We first initialize the precondition Φ to true. Then we invoke the predictive testing function *execute_program* (described in Section 3.1) on f and each test input vector TestInp in TestSuite to generate preconditions for the function f. The precondition returned by each invocation of *execute_program* is then added to Φ as a conjunct. *A precondition returned by execute_program has the following property: if the precondition is violated for some input, then on that input the function f results an erroneous execution.* Therefore, we can simply add a precondition as an assertion at the beginning of the function f. Observe that a precondition can always be violated for some inputs (otherwise, we can remove the precondition); however, this does not mean that the function has errors because in a program the function might be called in a context that avoids bad inputs to the function.

This way of generating a precondition using an existing unit test suite and adding it as an assertion at the beginning of the unit function significantly amplifies the effectiveness of predictive testing as follows. Suppose, we have a test input for the program P and on that input the execution of P calls f. Suppose, the execution path does not take a path inside f that has an assertion, say $\boldsymbol{a}$. Then PRETEX will not verify the assertion $\boldsymbol{a}$. However, if we have already encountered a path containing $\boldsymbol{a}$ inside f during unit testing, we have generated a precondition involving the assertion $\boldsymbol{a}$. Moreover, this precondition, being inserted at the beginning of f, will be encountered along the original execution of P and will be checked by PRETEX. In this way we verify the assertion $\boldsymbol{a}$ along the execution path inside f, even though we do not encounter the assertion along the original execution path of P.

3.3 Predictive Testing for Integer Overflow and Division by Zero

Any property that can be enforced as assertions inside a program can be checked predictively by PRETEX. In this section and subsequent sections, we show how we instrument assertions to detect various bugs including integer overflow or underflow, division-by-zero, memory safety, and string buffer overflows.

Let the type associated with m in a statement of the form $m \leftarrow e$ be τ. If τ is an integer type, that is, $\tau =$ `char` or `short` or `int` or `unsigned int` etc., then let the maximum value that can be taken by a variable of type τ be τ_{MAX} and the minimum value that can be taken by such a variable be τ_{MIN}. While executing a statement of this form, PRETEX symbolically evaluates e to a symbolic expression s and checks if the path constraint implies the assertion $\tau_{MIN} \leq s \leq \tau_{MAX}$. If it does not, then PRETEX predicts an integer overflow or underflow bug. Using a constraint solver, PRETEX generates a set of concrete inputs that causes the assertion to fail. It is worth mentioning that we maintain the constants in a symbolic expression in arbitrary precision integers to avoid underflow and overflow in a symbolic execution.

If the expression e in a statement `s` is of the form e_1/e_2, then PRETEX evaluates e_2 in the concrete state $\mathcal{M}$. If the result of the evaluation equals 0, then there is a division by zero bug in the execution that results from concrete execution on the test input provided. If it does not evaluate to 0 and the concrete test inputs themselves do not result in a divide by zero bug, then PRETEX checks that path constraint implies the assertion $\mathit{evaluate\_symbolic}(e_2) \neq 0$. If the assertion is not guaranteed by the symbolic path constraints, then PRETEX predicts a division by zero bug and generates a input vector that results in division by zero.

3.4 Predictive Testing for Memory Safety

For memory safety, we insert dynamic checks inside the program using a technique based on the backward-compatible memory-safety checking technique proposed in [12].

Whenever memory is allocated for an object, PRETEX saves in a table, called the *range table*, the lower bound

String function	Assertions and Updates
strcpy (dest,src)	src.null_pos > 0 dest.max_size ≥ src.null_pos dest.null_pos = src.null_pos

strncpy (dest,src,n)	src.null_pos > 0
	MIN(n, src.null_pos) ≤ dest.max_size
	if(n ≥ src.null_pos)
	if(dest.null_pos == 0 || dest.null_pos > src.null_pos)
	dest.null_pos = src.null_pos;
	else
	if (d.null_pos ≤ n)
	d.null_pos = 0;

strcat (dest,src)	src.null_pos > 0 dest.null_pos > 0 (src.null_pos + dest.null_pos -1) ≤ dest.max_size) dest.null_pos = (src.null_pos + dest.null_pos -1);
strncat (dest,src,n)	temp_src_size = MIN(n+1,src.null_pos) src.null_pos > 0 dest.null_pos > 0 dest.max_size ≥ (temp_src_size + dest.null_pos - 1) dest.null_pos = temp_src_size + dest.null_pos - 1
strchr (src,c) , strrchr (src,c)	src.null_pos > 0
strstr (s1,s2) , strpbrk (s1,s2) , strtok (s1,s2)	s1.null_pos > 0 s2.null_pos > 0
d=strdup(s)	s.null_pos > 0

Table 1: Assertions inserted to predict string related vulnerabilities

and the upper bound of the memory region that is allocated. This entry is deleted from the table when the object is deallocated. The bounds of the object's memory region are represented as unsigned long integers. Thus at any point during the execution, the *range table* contains a list of the bounds of all valid memory objects.

If there is a pointer dereference in an expression e of the form $*p$, then PRETEX evaluates p concretely as an unsigned long integer value and looks for an entry in the *range table* that contains p. If it does not find a corresponding range, that is, it does not find a pair of lower bound (lb) and upper bound (ub) in the *range table* such that $lb \leq p \leq ub$, then p refers to an invalid memory location. PRETEX reports this as an unsafe memory-access bug and aborts. If PRETEX succeeds in finding a range for p, then it evaluates p in the symbolic state $\mathcal{S}$. Let (lb, ub) be the range that it found in the previous step. It obtains the maps of the variables lb and ub in the symbolic state $\mathcal{S}$. PRETEX checks whether the path constraint implies the assertion *evaluate_symbolic*$(lb \leq p \leq ub)$. If it does not guarantee that the assertion is satisfied for possible inputs that take this path, then PRETEX predicts a memory access bug. A concrete input vector that causes this bug is generated by the constraint solver and returned by PRETEX to the tester.

PRETEX uses CIL [19] to instrument and simplify programs. CIL simplifies all array references and structure field references to simple pointer dereferences. Thus, PRETEX only needs to deal with simple pointer dereferences. Moreover, since CIL converts all local variables to have only function scope and block scope is removed, PRETEX doesn't need to manage entries in the *range table* when a block is entered or exited. PRETEX creates *range table* entries for global and static objects that exist for the whole program execution. Dynamically allocated objects, such as those allocated using `malloc()`, are added to the range table dynamically during the execution of the program and their corresponding entries are deleted whenever there is a call to `free()`. *Range table* entries for local variables are created at the beginning of a function definition and are deleted at the end of the function block.

In addition to pointer deferences, PRETEX also verifies the safety of pointer operations to insure spatial memory safety. If there is a pointer operation of the form, $r = p \pm i$, where r and p are pointers and i is an integer, then PRETEX ensures spatial memory-safety by checking that p and r both refer to the same *range table* entry. PRETEX obtains the lower bound of the object being pointed to by p and the lower bound of the object being pointed to by r after the pointer arithmetic operation. If the two lower bounds are not equal then PRETEX concludes that p and r do not point to the same object and it returns an error to the tester indicating a spatial memory-safety violation.

3.5 Predictive Testing for String Related Vulnerabilities

PRETEX detects a subset of buffer overflow bugs by symbolically tracking where the first guaranteed null character is within a string. We associate two utility variables `max_size` and `null_pos` with each program variable of type `char*` or `char[]`. `max_size` represents the number of bytes allocated in the buffer pointed to by the variable. `null_pos` represents the position of the first null character ('\0') in the buffer. For example, `char str[20]` has a `max_size` of 20. The `max_size` of an unbound pointer variable such as `char* str` is not set until an assignment statement, such as `str = "foo"` or `str = malloc(n)` is encountered. If a null ('\0') is guaranteed in the buffer at index i, then the `null_pos` for the buffer is set to $i + 1$. If there is no guaranteed null within the buffer, then `null_pos` represents this with the value 0. PRETEX instruments relevant assertions in the program P before each string function. A function call to `strcpy(dest,src)` is instrumented as follows:

```
assert(src.null_pos > 0 && dest.max_size ≥ src.null_pos);
strcpy(dest,src);
dest.null_pos = src.null_pos;
```

Note that `src` and `dest` are not structures. For convenience,

null_pos for src has been denoted as src.null_pos and its max_size as src.max_size.

Since a char buffer that is a program input may contain null ('\0') at any position in the buffer or may not include a null ('\0') at all, PRETEX maps null_pos to a symbolic variable in the symbolic state $\mathcal{S}$. This symbolic variable represents the set of all possible values that null_pos can assume for the char buffer it represents. The value of null_pos can only be non-negative and is always less than or equal to the buffer's max_size. PRETEX adds src.null_pos > 0 && dest.max_size $\geq$ src.null_pos to the path constraint to check that strcpy is being used safely. PRETEX evaluates the assertion in the concrete state $\mathcal{M}$ and if it evaluates to 0, then there is a bug that results from the concrete test case. If the test case itself does not cause a bug, then PRETEX evaluates the expression in the symbolic state $\mathcal{S}$ and verifies that the path constraint guarantees the expression. If it does not, then PRETEX uses a constraint solver to generate the concrete input values that cause a buffer overflow. Table 1 lists the assertions for all the string functions that PRETEX instruments.

4. IMPLEMENTATION AND EVALUATION

We have implemented predictive testing for C in a prototype tool called PRETEX. The symbolic execution engine of PRETEX is written in C. The front-end of PRETEX, which is written using the CIL [19] framework, inserts assertions into the source code to detect generic bugs and also inserts the function calls that perform symbolic execution. In the front-end, we transform the source code to three address code before instrumentation. For each instruction in three address code PRETEX looks at the type of lvalue in the set instruction, the type of the expression being assigned, and the function name if there is a function call. Based on these, PRETEX inserts the relevant assertions. PRETEX keeps track of the values of null_pos and max_size for all char buffers and tracks the bounds of all valid memory objects in hash table data structures. PRETEX uses the Yices [9] SMT solver to check satisfiability of formulas.

We applied PRETEX on several small and medium sized programs. We next present our experiences with gzip, bc and hoc calculators, and the space and the printtokens programs from the Software-artifact Infrastructure Repository (http://sir.unl.edu/portal/index.html).

Experimental Setup. Before running PRETEX, we had to merge all program source files into a single source file using CIL. PRETEX then performed instrumentation. We manually identified and tagged program inputs so that PRETEX can create symbolic values for such inputs and track symbolic constraints over them. We ran the PRETEX-instrumented program executables on several existing test inputs. Unfortunately, these programs do not come with user-defined assertions. Therefore, we ended up checking for generic bugs such as buffer overflow and integer overflow. We next report the results of our experiments.

Results. We ran the PRETEX-instrumented gzip (version 1.2.4) executable to compress and decompress several files. PRETEX was able to predict a known buffer overflow bug in gzip-1.2.4. In the function get_istat, strcpy (ifname, iname) copies the string iname to a char array ifname[1024]. The function treat_file calls get_istat by passing in the name of the file to be compressed or decompressed as the value for the parameter iname. PRETEX discovered that if the length of the filename is 1024 or more, the buffer ifname gets overrun. PRETEX discovered the bug by running gzip on a file whose filename has a length 10. The concrete execution missed this buffer overflow bug.

For bc (version 1.06), we used the test cases provided in the *bug* benchmark (http://www.ews.uiuc.edu/~chaoliu/sober/bc_input_1.tgz). For the hoc calculator, we manually generated test cases. PRETEX predicted no bug in these programs.

The space program from the Software-artifact Infrastructure Repository (SIR) acts as an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements, and checks the contents of the file for adherence to the ADL grammar and consistency rules. If the ADL file is correct, space outputs an array data file containing a list of array elements, positions, and excitations; otherwise, the program prints error messages. The program has a strcpy bug in its main function. A call to the strcpy function copies the name of the input file to a char buffer of size 31. By running a test case provided with the program, PRETEX predicted this buffer overflow even though the filename of its input file was not long enough to overflow the buffer.

The printtokens program from the Software-artifact Infrastructure Repository is a lexical analyzer. We modified this program so that its inputs include a variable that controls the size of the array token_str in the function get_token and value of the variable ind in the function get_actual_token. We ran PRETEX on this program with input values 80 and 79 for the size of token_str and ind, respectively. For these input values, the normal test execution detected no buffer overflow. PRETEX predicted buffer overflows in two strcpy calls: strcpy(token_ptr->token_string,token_str); in the function get_token and in the strcpy call strcpy(token_ptr->token_string,token_str); in the function error_or_eof_case.

We summarize the results of our experiments in Table 2. The first column specifies the name (and the version number) of the program that was tested using PRETEX. The second column gives the average execution time of the uninstrumented programs on the given test inputs. The third column gives the average execution time of the programs with dynamic checks for buffer overflows and integer overflows. The fourth column gives the average execution time of PRETEX on these programs. The fifth column gives the average number of calls to the theorem prover. The sixth column gives the number of bugs predicted by PRETEX. Finally, the last column gives the LOC (Lines of Code) ratio between the PRETEX-instrumented file and the uninstrumented file. This gives an estimate of how many checks we have inserted in the actual code.

Discussion. The table shows that a program slows down by a factor of at most 100X when we simply insert the checks for buffer overflow and integer overflow. This slowdown is before we perform symbolic analysis for PRETEX. The symbolic analysis of PRETEX further slows down the execution time by a factor of 1.5X-8X. The ratio of the instrumented code and the uninstrumented code in the last column indicates that the dynamic checks blows up code size significantly. We believe that by using some form of static analysis [18, 6], we can significantly reduce the number of dynamic checks and therefore, reduce the runtime

Program	Execution time (w/o instrumentation)	Execution time (w/o symbolic analysis)	Execution time (with PRETEX)	# of Theorem Prover Calls	# of Bugs Predicted	Ratio of LOC
gzip-1.2.4	0.350s	2.712s	18.328s (file size $\sim$ 0.5KB)	177	1	10.6
bc-1.06	0.148s	10.357s	47.930s	20	0	11.1
hoc	0.350s	2.300s	3.500s	83	0	16.7
space	0.269s	2.132s	6.497s	11	1	15.8
printtokens	0.750s	30.892s	4.3 min (file size $\sim$ 4KB)	3142	3	8.8

Table 2: Experimental results

overhead. This remains a future work. Observe that we invoke the theorem prover if the expression inside a check evaluates to a symbolic expression; if the evaluated expression has no symbol, then we skip theorem proving. In our experiments, we observed that the theorem prover is not invoked at every place where we have a check. Therefore, we do not get significant amount of slowdown when we perform side-by-side symbolic analysis. Our experiments also establish the fact that PRETEX can predict bugs that do not occur in the actual test execution, but could occur in a "nearby" execution.

5. CONCLUSION

An attractive feature of PRETEX is that PRETEX integrates naturally with existing testing methodologies and software development cycles by preserving the existing testing interface that testers are already accustomed to. By selectively limiting the path-space that is searched for each test run, our tool verifies a small but important subset of the path-space and provides immediate results. PRETEX leverages already existing test cases and regression tests in order to discover new bugs that were previously undiscovered.

6. REFERENCES

[1] T. Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, MSR, 2003.

[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. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. of SPIN Workshop*, 2005.

[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] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low-level programs. Technical Report EECS-2006-129, UC Berkeley, 2006.

[7] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[8] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *27th International Conference on Software Engineering*, 2005.

[9] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for dpll(t). In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 81–94, 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] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *AADEBUG*, 1997.

[13] 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.

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

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

[16] E. Larson and T. Austin. High coverage detection of input-related security faults. In *Proc. of the 12th USENIX Security Symposium (Security '03)*, Aug. 2003.

[17] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE. (To Appear).

[18] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *TOPLAS*, 27(3), May 2005.

[19] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and transformation of C Programs. In *Proceedings of Conference on compiler Construction*, pages 213–228, 2002.

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

[21] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *19th European Conference Object-Oriented Programming*, 2005.

[22] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.

[23] M. L. Soffa, A. P. Mathur, and N. Gupta. Generating test data for branch coverage. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 219. IEEE Computer Society, 2000.

[24] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 253–262, 2005.

[25] 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.

[26] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *17th IEEE International Conference on Automated Software Engineering*, 2002.

[27] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Procs. of TACAS*, 2005.