LATEST : Lazy Dynamic Test Input Generation

Rupak Majumdar
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-36
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-36.html

March 20, 2007

LATEST: Lazy Dynamic Test Input Generation

Rupak Majumdar
UCLA
rupak@cs.ucla.edu

Koushik Sen
UC Berkeley
ksen@cs.berkeley.edu

Abstract

We present *lazy expansion*, a new algorithm for scalable test input generation using directed concolic execution. Lazy expansion is an instantiation of the counterexample-guided refinement paradigm from static software verification in the context of testing. Our algorithm works in two phases. It first explores, using concolic execution, an abstraction of the function under test by replacing each called function with an unconstrained input. Second, for each (possibly spurious) trace generated by this abstraction, it attempts to expand the trace to a concretely realizable execution by recursively expanding the called functions and finding concrete executions in the called functions that can be stitched together with the original trace to form a complete program execution. Thus, it reduces the burden of symbolic reasoning about interprocedural paths to reasoning about intraprocedural paths (in the exploration phase), together with a localized and constrained search through functions (in the concretization phase).

Lazy expansion is particularly effective in testing functions that make more-or-less independent calls to lower level library functions (that have already been unit tested), by only exploring relevant paths in the function under test. We have implemented our algorithm on top of the CUTE concolic execution tool for C and applied it to testing parser code in small compilers. In preliminary experiments, our tool, called LATEST, outperformed CUTE by an order of magnitude in terms of the time taken to generate inputs, and in contrast to CUTE, produced many syntactically valid input strings which exercised interesting paths through the compiler (rather than only the parser error handling code).

1. Introduction

Automatic, *complete*, and *scalable* test input generation has been an important research problem in software engineering for a long time. Over the years, several different test generation techniques have been suggested that satisfy a subset of the above properties. For example, *random testing* chooses the values over the domain of potential inputs randomly [3, 8, 7, 20, 21]. The problem with random testing is twofold: 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 [19]. Thus, while automatic and scalable, random testing is rarely complete. In contrast, specification-guided manual test generation can pick the right set of tests and achieve high coverage (completeness), but it is neither automatic, nor scalable. The search for techniques that achieve all three properties is important, not only academically but also economically, as the major cost of software development today is in testing and validation.

Testing is a *dynamic* analysis technique. By running the program on concrete inputs, testing is constructing an under-approximation of the set of reachable program states. *Static analysis* provides a dual view: instead of running the program on concrete inputs, one algorithmically simulates running the program over some abstract space of constraints, so that the abstractly reachable states are guaranteed to contain the concretely reachable states. Static analysis is automatic and complete, and many recent implementations show it is scalable (for many abstractions of practical interest) to millions of lines of code. Unfortunately, it is *imprecise*: unlike testing, which only exercises feasible executions, static analysis can return errors or warnings even if the code is error free, because it has lost too much information in abstraction.

Clearly, testing and static analysis have complementary strengths. Is it then possible to combine the two in hybrid algorithms that can amplify the advantages of both, while alleviating their individual disadvantages?

This is also a well-studied question. One instantiation, symbolic execution [14, 6, 26, 27], has been applied to test generation since the 70's. In pure symbolic execution, a program is executed using symbolic variables in place of concrete values for inputs. Each branch statement in the program generates a symbolic constraint that determines an *execution path*. The goal is to generate concrete values for inputs which would result in different paths being taken. Completeness is achieved by depth first exploration of the paths using backtracking [13]. While in theory automatic and complete, for large or complex programs, symbolic execution is expensive, and worse, imprecise as complex program behaviors cannot always be predicted statically, and these techniques have been limited in their application.

To address the limitations of pure symbolic execution, we have recently proposed concolic testing [10, 23, 16] based on dynamic methods for test input generation [15, 25]. Concolic testing iteratively generates test inputs by combining concrete and symbolic execution, observing that the complexity and imprecision of purely symbolic techniques can be alleviated by using concrete values from random executions. During a concrete execution, a conjunction of symbolic constraints placed on the symbolic input variables along the path of the execution is generated. These constraints are modified and then solved, if feasible, to generate additional test inputs which would direct the program along alternative paths. Specifically, conjuncts in the path constraint are systematically negated to generate a set of inputs that provides a depth first exploration of all paths. If it is not feasible to solve the modified constraints, then simply substitute random concrete values.

Concolic testing shows how static and dynamic program analyses can be combined to automatically and completely generate test inputs. Unfortunately, concolic testing does not scale to large programs, because the number of feasible execution paths of a program increases exponentially with the increase in the length of an execution path. The problem is exacerbated in the presence of function calls. A sequence of n calls to a function with k paths already leads to k^n distinct paths to be explored. We call this the *path explosion problem.*

Static analysis techniques deal with the path explosion problem using several techniques. There are a couple of key insights in these techniques that can be used. First, perform *interprocedural analysis* by memoizing *summaries* (input-output behaviors) for functions, so that subsequent calls to the function can simply look up the summary rather than explore all paths of the function [24, 22].

Second, always explore the most abstract version of the program, refining the abstraction on demand based on particular executions ("counterexample-guided refinement") [5, 1, 12]. These techniques are orthogonal, and complementary. Concolic execution with summarization was studied recently [9], and found to have a dramatic increase in scalability. In this paper, we propose a novel technique similar to counterexample-guided refinement to address the path explosion problem in concolic testing.

In counterexample-guided refinement (CEGAR) [5, 12], one starts with a coarse abstraction of the program. The algorithm then uses counterexample paths (i.e., abstract paths to an error state) to either find concrete executions (if the abstract path is realizable concretely) or to find refinements of the abstraction (if the abstract path is not feasible). This process is repeated until either there is a test case that reaches an error state, or there is a proof that the error states are not reachable.

Counterexample-guided refinement has been used for test generation before [2]. However, the previous technique had several shortcomings. First, abstract exploration is still orders of magnitude slower than program execution. Second, good abstract domains are currently known only for control-dominated programs. For applications with complicated data flow, abstract exploration fails to produce test cases leading to interesting program points. Third, the refinement phase was performed purely symbolically [12], inheriting all the imprecisions of symbolic execution. (A recent implementation of CEGAR uses concolic execution for refinement [11].) In this paper, we instead explore a *fully dynamic implementation* of the CEGAR loop, where both abstract path exploration and refinement are driven by concolic execution.

We assume that we are interested in complete exploration of a function-under-test, which makes calls into a library that has already been unit tested. Thus, we are interested in complete coverage of the top level function, and want to explore only the paths in the library that lead to observably different behaviors in the top level. We assume we also have the code for the library.

Our algorithm, called *lazy expansion*, works as follows. We explore the function under test using concolic testing, but replace the return value of each library call, whose behavior depends on a program input, with an unconstrained input. This creates an abstraction of the program: by ignoring data correlations in the library calls, we can generate paths that are not concretely realizable. However, the abstraction reduces the complexity of interprocedural path exploration to *intraprocedural* path exploration. Given such a path π in the abstraction, we perform *path refinement*, that is, expand the called functions along the path to get a concretely realizable path whose projection on the function under test is π. Path refinement performs a backtracking search over the path π, trying to find concrete paths through each called function that can be stitched together. The expansion of functions recursively invokes the lazy expansion algorithm, *lazily* expanding functions along the path. By expanding the paths inside the nested functions on demand, and under the constraints of the overall path in the top level function, we only explore relevant parts of the program path space. This significantly prunes our search while retaining the relative soundness and completeness of concolic testing.

We have implemented the lazy expansion algorithm in a tool called LATEST, built on top of CUTE, a concolic unit testing engine for C [23]. We applied LATEST to generate test inputs for programs, most of which are parsers. Parser programs are especially suited for our technique. The parser makes many almost independent calls to the lexer. The lexer has many different paths through it. However, lazy expansion ensures that a minimal number of paths through the lexer gets explored. In our experiments, we find that LATEST outperforms CUTE by several orders of magnitude. For example, on a parser for PL/0, CUTE never managed to generate valid PL/0 programs after running for one day; however, LATEST generated all 99 PL/0 programs having 8 tokens within 5 hours. We believe that through the novel interpretation of static analysis technique in a dynamic context, the LATEST algorithm takes an important step towards automatic, scalable, and complete test generation for large software systems.

```
int strcmp(char *str1, char *str2) {
    while (*str1 != '\0'
                  && *str2 != '\0'
                  && *str1 == *str2) {
        str1++; str2++;
    }
    return *str1 - *str2;
}

void testme() {
1:  int tmp1, tmp2;
2:  char *s1, * s2;

3:  char *const1 = "Hello World";
4:  char *const2 = "Hello ESEC/FSE";

5:  s1 = input();
6:  s2 = input();

7:  tmp1 = strcmp(s1,const1);
8:  tmp2 = strcmp(s2,const2);
9:  if (tmp1 == 0) {
10:     if (tmp2 == 0) {
11:        printf("Success");
        }
    }
}
```

Figure 1. Example

2. Motivating Example and Overview

We motivate our lazy expansion test generation algorithm using a small example.

The Problem. Consider the example in Figure 1. We want to unit test the function `testme`. The function has two inputs `s1` and `s2`, which can get any null-terminated string from an external environment. We use the statement `v = input();` to indicate that the variable `v` is assigned an external input. The function `testme` compares `s1` with the constant string `const1="Hello World"` (line 7) and `s2` with the constant string `const2="Hello ESEC/FSE"` (line 8). If both checks return 0 (indicating that the strings `s1` and `s2` are respectively equal to the strings `"Hello World"` and `"Hello ESEC/FSE"`, the statement at line number 11 is reached.

The function `strcmp` performs the string comparison operations. It takes two null-terminated strings `str1` and `str2` as input, and walks down the string buffers, comparing character by character until it reaches the first character at which either the strings differ or one or both of them reach the last null character '\0'. At this point, the difference between the current characters is returned. A difference of 0 indicates that the strings are the same.

We assume that we have already performed unit testing of the function `strcmp` and the function has no bug.[1] Our goal is to perform unit testing of the function `testme`. In particular, we want to get *path coverage* of `testme`. We focus on test input generation for `testme` that will cover all feasible execution paths of

[1] Note that the function `strcmp` is not bug-free: it crashes if `str1` or `str2` are `NULL`; however, to keep the function simple we ignore these cases and assume that the arguments to the function `strcmp` are always null-terminated.

testme. A feasible execution path of testme is a path (sequence of assignments, conditionals, and function call statements) in its control flow graph that can be executed for some setting of the inputs s1 and s2.

The function testme has 720 distinct execution paths and these paths can be executed by setting the values of the inputs s1 and s2 suitably. The number 720 is derived as follows. The function strcmp with an input string and a constant string (i.e., where the argument str1 is an input and the argument str2 is a constant string) has $2(n+1)$ distinct execution paths, where n is the length of the constant string. Therefore, the first invocation of strcmp (line 7) can result in 24 distinct execution paths and the second invocation of strcmp (line 8) can result in 30 distinct execution paths. However, the return values from the two calls to strcmp determine the path exercised in testme. We can combine each execution path through the first invocation of the strcmp function with each execution path through the second invocation of the function strcmp to get a distinct path in the function testme. Therefore, there are $30 * 24 = 720$ execution paths in the function testme. A dynamic test input generation tool based on concolic testing, such as DART, CUTE, or EXE [10, 23, 4], will generate 720 distinct inputs to perform unit testing of this function. We verified this fact by invoking CUTE on this code.

However, if we overlook the paths inside the strcmp function, then the function testme has only 3 paths. Therefore, to successfully unit test the function testme, assuming that strcmp has been separately tested, rather than generating 720 test inputs, we should ideally generate 3 sets of values for the inputs s1 and s2 on which the function will execute each of these 3 paths exactly once. This would be possible if the function strcmp could return any possible value (i.e., acted as an input). In general, though, one would have to search through the possible executions of strcmp to find paths that return appropriate return values for the 3 paths to be executed in testme. This search can often be performed independently for each call to strcmp.

Our *lazy dynamic test input generation* algorithm, called LATEST, is based on this observation. In particular, we observe that a number of paths of the function strcmp have the same effect on the outcome of the conditional statements at line 9 and line 10 of the function testme. In fact, for the purpose of testing the function testme, all the paths through the function strcmp can be partitioned into two sets: one set of paths that returns a non-zero value and the other set of paths that returns a zero value. Therefore, to efficiently test the function testme, we need to consider only one execution path from each partition (or equivalence class). This key insight motivates our LATEST algorithm.

The LATEST Algorithm. The LATEST algorithm lazily expands called functions when testing a unit. Conceptually, the algorithm has two phases: an initial *abstract exploration* phase, followed by a *concretization* phase. Here, we describe the algorithm under the assumption that there are no globals and functions do not have side effects. We shall lift this restriction in Section 4.

The abstract exploration phase systematically explores paths of the function under test, but assumes that every called function can return any possible value, that is, abstracts all called functions into unconstrained external inputs.

In our example, when expanding testme, the nested function calls to strcmp are abstracted to return any value. The pseudo-code for the abstraction is given in Figure 2. We have removed the calls to the function strcmp, and assign inputs to tmp1 and tmp2. This creates an *abstraction* of the function testme: the set of paths explored in this abstraction is a superset of the concretely executable paths, however, some paths can arise because of the imprecision in modeling the called function and may not be concretely feasible.

We generate abstract execution paths of the abstracted function using *concolic execution* [10, 23, 4]. In this technique, the function is run simultaneously with concrete inputs as well as with symbolic inputs. Along the execution, the symbolic part gathers constraints on the program inputs that cause execution to follow the current path. The symbolic constraints are simplified using the concrete values that are available during execution. As a result of concolic execution, each point of the executed path is annotated with (1) the concrete store at that point, (2) the symbolic store that maps addresses to symbolic expressions, and (3) a path constraint that tracks symbolic constraints for the conditionals executed along the path to reach the point. At any branch point, the path constraint can be modified so that any input satisfying the modified constraints will cause the corresponding program execution to follow the branch of the conditional opposite to the one followed in the current execution. In this way, all paths of the abstract function can be explored.

In the abstraction of testme from Figure 2, concolic execution explores the three execution paths by generating the following sets of values for tmp1 and tmp2: {tmp1=1, tmp2=5}, {tmp1=0, tmp2=5}, and {tmp1=0, tmp2=0}.

The explored paths are abstract because they are obtained by concolic testing of the abstraction of testme. These paths may not be feasible in practice: one has to "stitch together" paths from the called function strcmp in order to get a feasible execution with the same branching behavior in testme. This is the responsibility of the *concretization* phase, which expands the called functions to find concrete paths through them.

In order to do this, the $concretize$ function uses the symbolic stores generated by the concolic execution while exploring the path as well as the path constraint for the (abstract) path. For each function call $g()$ along the path, $concretize$ performs systematic path exploration through the called function g starting from the symbolic store at that point in the path. The concolic execution is used to find a path π through g such that the path constraint of the top level function is consistent with the path constraint of π. This ensures that this path can be stitched together with the abstract path in the top level function. Notice that the same abstract exploration followed by concretization can be recursively applied while exploring g, thus, transitively called functions are lazily expanded in the search for paths.

If several functions are called in sequence along the path, $concretize$ implements a backtracking search through the paths of the functions until it finds concrete paths in each of the functions that can be feasibly stitched together. Of course, this may not always be possible, in which case it returns that the abstract path cannot be concretized.

The LATEST algorithm calls the abstract exploration and concretization routines in a loop until there are no more abstract execution paths to explore. Each concretizable path gives a test input. The concretized paths are used to update coverage goals in the function under test.

Application to testme. In testme, we lazily expand the function calls to strcmp for each of the three paths as follows. Consider the abstract path

$$(5; 6; 7; 8; 9; 10; 11)$$

that we obtain by executing testme in Figure 2 on the input {tmp1=0, tmp2=0}. The constraint on the variables tmp1 and tmp2 for this path is

$$(\texttt{tmp1} = 0) \wedge (\texttt{tmp2} = 0) \quad (1)$$

Given this abstract path constraint, we try to find concrete paths in the two invocations of strcmp (on lines 7 and 8) so that the values returned by these invocations satisfy the abstract path constraint.

In order to concretize the abstract path, we start concolic exploration of the paths inside the first invocation of strcmp under

```
void testme() {
    ...
3:  char *const1 = "Hello World'';
4:  char *const2 = "Hello ESEC/FSE'';
5:  s1 = input();
6:  s2 = input();

7:  tmp1 = input();
8:  tmp2 = input();
9:  if (tmp1 == 0) {
10:     if (tmp2 == 0) {
11:        printf("Success");
        }
    }
}
```

Figure 2. Abstraction of `testme`

the actual context (in the actual context the argument `str1` denotes the input `s1` and the argument `str2` is set to the string `"Hello World"`). After generating 24 input values for `s1`, we find a path inside the first invocation of `strcmp` that satisfies the abstract path constraint $(\texttt{tmp1} = 0) \wedge (\texttt{tmp2} = 0)$. This path is obtained when `s1 = "Hello World"`. At this point, the path is partially expanded: the first call to `strcmp` has been concretized, but now we have to find a path through the second call. We update the path constraint to be the conjunction of the path constraint from (1) with the path constraint generated by the path inside `strcmp` (which is `s1[0]='H'` $\wedge$ `s1[1]='e'` $\wedge \ldots \wedge$ `s1[12]='\0'`).

Next, we try to find a path inside the second invocation of `strcmp` that satisfies this updated abstract path constraint. Such a path is obtained by the concolic exploration of the `strcmp` in the actual context. The path is obtained after generating 30 inputs. For this path concolic exploration generates `s2="Hello ESEC/FSE"`. Finally, we generate the actual inputs `s1 = "Hello World"` and `s2="Hello ESEC/FSE"` for which the function `testme` takes the desired path. This path is generated after 55 iterations (i.e., after exploring 55 paths.)

In a similar way, for the other two abstract paths, we generate values for `s1` and `s2` by expanding the invocations of the function `strcmp`. These inputs are generated after 26 and 32 iterations respectively. In total, we perform 113 iterations to obtain complete path coverage in the `testme` function only. This number is significantly less than 720.

Notice that each concolic execution is only intraprocedural: nested function calls are expanded only in the concretization phase. This novel way of exploring the nested functions on demand enables LATEST to avoid unnecessary exploration of the nested functions, which we have already unit tested.

Our technique is similar to inter-procedural exploration with summarization [22], recently suggested as an improvement to concolic execution [9]. However, our requirements are weaker (only test the top level function, assuming the callees have been unit tested), and so we can optimize the exploration further by only constructing summaries for relevant paths in the top level functions. Without this assumption, the technique of [9] has to comprehensively explore all paths in the callees as well. Of course, these techniques are complementary: in the exploration of called functions, we can summarize states and use the summaries in subsequent explorations. However, we leave out this optimization in our description and experimentation as it is orthogonal to the technique of the paper, and also confounds the benefits of our approach with the benefits of summarization in the experimental results.

Motivation from a Real-World Test Generation Problem. The technique behind LATEST was primarily motivated due to the inability of DART or CUTE to generate valid inputs for parsers. A parser usually invokes a lexer function, say `lex`, several times along a path. The `lex` function in turn may call `strcmp` function or similar regular expression matching functions several times. A simple skeleton of such a parser is shown in Figure 3. In practice, a simple `lex()` functions can have many paths (for example > 10000.) Therefore, DART or CUTE requires to generate 10000^5 inputs to even generate valid programs with 5 tokens for the parser. In summary, the problem of test input generation for parsers and similar programs becomes intractable due to the *path-space explosion*. A recently developed technique called SMART, tries to address this path explosion problem, by summarizing functions. However, a summary for complex function such as `lex()` can result in a huge disjunctive formula. Our experience shows that a conjunction of such huge summaries usually makes the constraint solving problem intractable.

```
Token lex() {

  istream = input();
  ...
  if (strcmp(istream,"while")==0) {
     // return WHILE token
  }
  ...
  if (strcmp(istream,"else")==0) {
     // return ELSE token
  }
  ...
}

parse() {
   Token token;

   while (token = lex()) {
     // do something with token
   }
}
```

Figure 3. Example

LATEST tries to address this problem by employing a demand-driven and sound strategy for exploring paths. For a parser, LATEST delays the expansion of the functions `lex` and `strcmp`. LATEST abstracts those functions using unconstrained inputs. The abstract parser function is then explored using concolic execution. Suppose, LATEST gets 100 abstract paths in the parser function with say 5 tokens. Then for each such abstract path, LATEST tries to expand the 5 invocations of the `lex` function. Assuming that the `lex` function has at most 10000 execution paths, LATEST usually ends up expanding the 5 invocations of `lex` in less than $5 * 10000$ iterations. Finally, LATEST completes the complete path exploration of the parser function with 5 tokens in $5 * 10000 * 100$ iterations which is exponentially less than 10000^5 iterations. Note that during the exploration of the paths inside `lex`, LATEST abstracts the invocation of the functions `strcmp`. This way LATEST reduces the number of iterations recursively.

3. Abstract Concolic Exploration

Our LATEST algorithm has two phases: an *abstract exploration* phase, followed by a *concretization* phase. The abstract exploration phase uses concolic testing to explore all paths of a function in which all function calls has been abstracted. We call this *abstract concolic exploration*. We next describe the abstract concolic exploration algorithm in terms of the function $abstract\_next\_path$. The $abstract\_next\_path$ function is almost similar to concolic execution [23, 10], except the fact that in concolic execution we do not

abstract function calls. To keep the paper complete and to clarify the differences we give the definition of $abstract\_next\_path$.

3.1 Programs and Concrete Semantics

We fix an imperative programming language to illustrate our abstract concolic exploration algorithm as well as our LATEST algorithm. The operations of the programming language consist of labeled statements $\ell : s$, where labels correspond to the program counter. A statement is either (1) an *input statement* $\ell : m := \mathsf{input}()$ that reads an external input into the lvalue m, (2) an *assignment* statement of the form $m := e$, where m is an lvalue and e is a side-effect free expression, (3) a *conditional* statement of the form $\mathsf{if}(e)\mathsf{goto}\ \ell$, where e is a side-effect free expression and ℓ is a program label, (4) a *function call* of the form $m := f(m_1, \ldots, m_n)$, for lvalues m, $m_1, \ldots, m_n$, or (5) a return statement of the form `return ret` for a special *return variable* `ret` used to pass back return values.

Execution begins at the program counter ℓ_0. For a labeled assignment statement $\ell : m := e$ or input statement $\ell : m := \mathsf{input}()$ we assume $\ell + 1$ is a valid program counter, for a labeled conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ we assume both ℓ' and $\ell+1$ are valid program counters, and for a function call $m := f(\ldots)$, we assume that the function f is defined and starts at a valid program counter ℓ_f, and additionally, $\ell + 1$ is a valid program counter.

The set of *data values* consists of program memory addresses and integer values. The semantics of the program is given using a *memory* $\mathcal{M}$ consisting of a mapping from program addresses to values. Execution starts from the initial memory $\mathcal{M}_0$ which maps all addresses to some default value in their domain. Given a memory $\mathcal{M}$, we write $\mathcal{M}[m \mapsto v]$ for the memory that maps the address m to the value v and maps all other addresses m' to $\mathcal{M}(m')$. We assume that the concrete semantics of the program is implemented as a function $eval\_concrete$ that takes a memory and an expression, and evaluates the expression in the memory $\mathcal{M}$. Additionally, we assume that a function $\mathsf{stmt\_at}(\ell)$ returns the statement with program counter ℓ.

Statements update the memory. The concrete semantics of the program is given in the usual way as a relation from program counter and memory to an updated program location (corresponding to the next instruction to be executed) and updated memory [17]. For an assignment statement $\ell : m := e$, this relation calculates, possibly involving address arithmetic, the address m of the left-hand side, where the result is to be stored. The expression e is evaluated to a concrete value v in the context of the current memory $\mathcal{M}$, the memory is updated to $\mathcal{M}[m \mapsto v]$, and the new program location is $\ell + 1$. For an input statement $\ell : m := \mathsf{input}()$, the transition relation updates the memory $\mathcal{M}$ to the memory $\mathcal{M}[m \mapsto v]$ where v is a nondeterministically chosen value from the range of data values, and the new location is $\ell + 1$. For a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$, the expression e is evaluated in the current memory $\mathcal{M}$, and if the evaluated value is zero, the new program location is ℓ' while if the value is non-zero, the new location is $\ell + 1$. In either case, the new memory is identical to the old one. For a function call $\ell : m := f(m_1, \ldots, m_n)$, the values of the actual arguments $\mathcal{M}(m_1), \ldots, \mathcal{M}(m_n)$ are copied to the formal parameters of f, and the return address $\ell + 1$ and the return lvalue m are pushed onto a stack. The new location is ℓ_f, the start location for function f. At a return statement `return ret` for a function f when the top of the program stack is $\langle \ell, m \rangle$, the new location is ℓ (function return to caller), and the memory is updated to $\mathcal{M}[m \mapsto \mathcal{M}(\mathtt{ret})]$, reflecting the copy-back of the return value. If the program stack is empty, the execution terminates.

Let L_{A} be the set of labels of assignment and input statements, L_{C} be the set of labels of the conditional statements, and L_{F} the set of labels of function call and return statements of P. An *execution path* w of P is a finite[2] sequence in $\mathbf{Execs} := L^*$, where $L = L_{\mathsf{A}} \cup L_{\mathsf{C}} \cup L_{\mathsf{F}}$ is the set of statement labels of P.

The concrete semantics of P at the RAM machine level allows us to define for each input map IMap an execution path: the result of executing P on IMap. Let $\mathbf{Execs}(P)$ be the set of such execution paths generated by all possible IMap. Note that the execution of P on several input maps may result in the same execution path. The set $\mathbf{Execs}(P)$ defines a *computation tree*, which is a rooted directed tree with nodes corresponding to program labels with root ℓ_0. A node ℓ corresponding to an assignment statement, a function call statement, or a return statement has one successor $\ell + 1$; a node corresponding to a conditional statement has two successors for the then and the else branches. The leaves of the computation tree correspond to `return` statements.

3.2 Abstract Concolic Execution

The pseudo-code of the $abstract\_next\_path$ function is given in Figure 4. This function takes the following as inputs: a function f whose abstract paths we want to explore, an initial symbolic store $\mathcal{S}_0$, and an initial symbolic path constraint ϕ. The initial symbolic store and the initial path constraint gives the symbolic context in which we want to explore f. This is essential for LATEST because in LATEST we will use $abstract\_next\_path$ to explore paths in nested functions under different contexts. If we want to explore f in an empty context then $\mathcal{S}_0$ should be an empty map and ϕ should be *true*. The function $abstract\_next\_path$ returns an abstract path in the function f and the path constraint along that abstract path. $abstract\_next\_path$ behaves like an *iterator*, which on every invocation returns a new abstract path in f that was not returned before. Any input, say IMap', that satisfies ϕ gives a valid path in f. For an input IMap', the initial concrete state is obtained by replacing all symbolic variables in $\mathcal{S}_0$ using the input map IMap'.

Like an iterator, $abstract\_next\_path$ has to maintain some persistent state across calls to it. This persistent state comprises of the next input map IMap, a concise history $path$ of the paths that we have so far explored, and a flag $completed$ indicating whether our exploration is over. We make these variables **static** in the function $abstract\_next\_path$. (Actually, $abstract\_next\_path$ needs to maintain one set of static variables for each function f and each context $\mathcal{S}$ and ϕ. We show only one set of static variables for readability.)

The function $abstract\_next\_path$ initializes a number of local data structures as follows: the symbolic store $\mathcal{S}$ is initialized to $\mathcal{S}_0$; a map IMap' from the symbolic values to concrete values is obtained by solving ϕ and the map is used to obtain the initial concrete memory $\mathcal{M}$ from $\mathcal{S}$; the program counter pc is initialized to the label ℓ_f of the first statement of f; the variables i and k denoting the number of inputs and the number of conditionals, respectively, encountered so far in the execution are initialized to 0; a sequence $path\_c$ of symbolic constraints generated so far along the execution is initialized to the empty sequence; a sequence π recording the annotated trace is initialized to the empty sequence. The annotated trace, which is a sequence of pairs of symbolic stores and statements, is returned by the function $abstract\_next\_path$. The LATEST algorithm uses this annotated trace for concretization as described in Section 4.

The functions $abstract\_next\_path$ returns $\mathsf{NoMorePath}$ if $completed$ is true, i.e., we have explored all the abstract paths in the function f. Otherwise, the functions $abstract\_next\_path$ executes the statements of f in a loops as follows. It computes the next statement $\mathbf{s}$ to be executed by calling $\mathsf{stmt\_at}(pc)$ and appends

[2] We thus assume that all program executions terminate; in practice, this can be enforced by limiting the number of execution steps.

(where ^ is the append operator) the pair $(\mathcal{S}, \mathbf{s})$ of the current symbolic store and the current statement to the annotated trace π. If **s** is an assignment statement, then the right hand side expression is evaluated both concretely and symbolically and the results are used to update the left hand side memory location in both the concrete state and symbolic store, respectively. The program counter pc is incremented by 1 to point at the next instruction. As an optimization, if the symbolic expression is a concrete value, then we drop m from the domain of the symbolic store. Thus, we maintain that all addresses in the domain of $\mathcal{S}$ are not concrete values.

If **s** is a conditional statement, then the predicate inside the conditional is evaluated both concretely and symbolically to get b and c, respectively. If b is true, then c is appended to the path constraint $path\_c$; otherwise, $\neg c$ is appended to $path\_c$. If k, the number of conditionals executed so far, is equal to the size of $path$, then 0 is appended to $path$ to record that the current branch needs to be negated in future to generate a new input that would force the program along a new unexplored execution path. Finally, k is incremented since we have executed a conditional.

If **s** a function call statement of the form $m := g(m_1, \ldots, m_n)$, then we abstract the return value of the function g by an input. We check if we have a value of the input available in the IMap (by checking if $\mathsf{IMap}[i]$ is defined). If not, we initialize $\mathsf{IMap}[i]$ with a random number as in DART and CUTE. We use the value $\mathsf{IMap}[i]$ to update the mapping of m in the concrete memory. We also create a fresh symbolic value s_i and use it to update the mapping of m in the symbolic store. We perform the same steps if **s** is an input statement. Finally, we increment the pc and the i.

Once we have reached the return statement of f, we terminate the execution and invoke $solve\_constraint$ to generate a new input that would force execution of f along a different path at the next call of $abstract\_next\_path$. To do so, we find the last branch that has not been negated before, say $path\_c[j]$, and generate a new IMap by solving $\phi \wedge path\_c[0] \wedge \ldots \wedge \neg path\_c[j])$. The entry $path[j]$ is update to 1 to indicate that the same branch should not be negated in a future execution.

If we do not manage to generate a new input then $completed$ is set to **true**. At the end, we return the pair $(\pi, path\_c[0] \wedge \ldots \wedge path\_c[k-1])$ containing the current annotated trace and the current path constraint inside the function f.

We use the function *evaluate_symbolic* to evaluate an expression symbolically in the symbolic store. Due to space constraints, we do not define this function here; however, interested readers can refer to [10, 23] for further details. An important feature of our symbolic evaluator is the following. Since the concrete values stored in all memory addresses are available at the time of symbolic evaluation, the symbolic evaluator can "fall back" on concrete values if either the expressions get too big, or the constraints go beyond the purview of the underlying constraint solver. For example, the tools described in [10] implement a solver for the theory of integer linear constraints. When an expression falls outside the theory, as in the multiplication of two non-constant sub-expressions, the symbolic evaluator simply falls back on the concrete value of the expression, which is used as the result. In such cases, we set a flag to **false** to indicate that our search algorithm can no longer be complete, i.e., we cannot explore all feasible abstract paths in f.

```
abstract_next_path f S_0 φ
    pc := ℓ_f
    S := S_0
    let IMap' satisfies φ in
        obtain M from S using IMap'
    i := k := 0
    path_c := π := empty sequence
    static IMap := empty map
    static path := empty sequence
    static completed := false
    if completed then
        return (NoMorePath, ·)
    s := stmt_at(pc)
    π := π ^ (S,s)
    while (s ≠ return ret)
        match (s)
            case (m := e):
                S := S[m ↦ evaluate_symbolic(e, M, S)]
                M := M[m ↦ evaluate_concrete(e, M)]
                pc := pc + 1
            case (if (e)goto ℓ'):
                b := evaluate_concrete(e, M)
                c := evaluate_symbolic(e, M, S)
                if b then
                    path_c := path_c ^ c
                    pc := ℓ'
                else
                    path_c := path_c ^ (¬c)
                    pc := pc + 1
                if |path| = k then
                    path := path ^ 0
                k := k + 1
            case (m := input()):
            case (m := g(m_1, ..., m_n)):
                // abstract the function. will be expanded by concretize
                if IMap[i] not defined then
                    IMap[i] := random()
                M := M[m ↦ IMap[i]]
                S := S[m ↦ s_i]
                i := i + 1
                pc := pc + 1
        s := stmt_at(pc)
        π := π ^ (S,s)
    (IMap, path, completed) := solve_constraint φ path_c path k
    return (π, path_c[0] ∧ ... ∧ path_c[k − 1])

solve_constraint φ path_c path k
    j := k − 1
    while (j ≥ 0)
        if (path[j] = 0) then
            if (∃IMap' that satisfies φ ∧ path_c[0] ∧ ... ∧ ¬path_c[j]) then
                path[j] := 1
                return (IMap', path[0..j],false)
            else j := j − 1;
        else j := j − 1;
    return (·, ·, false) // complete search is over
```

Figure 4. Abstract Concolic Exploration

4. Lazy Expansion

We now describe the LATEST algorithm using the $abstract\_next\_path$ function as a building block. We give a recursive formulation of the algorithm. The key point is that we always use abstract concolic execution implemented in $abstract\_next\_path$ to enumerate over intra-procedural abstract executions (i.e., executions where called functions are not followed but assumed to return any value), and use a special $concretize$ method to stitch together the intra-procedural paths to form an inter-procedural path.

4.1 Lazy Expansion: Conceptual Algorithm

We first illustrate the algorithm assuming that functions have no side-effects and always return a single value of primitive type. We shall lift this restriction in the next subsection.

Figure 5 shows the organization of the code. The main LATEST algorithm takes as input a function under test f and systematically tests it by generating abstract execution paths in f using $abstract\_next\_path$ (with the arguments f, an initial empty

```
LATEST f =
  while (true)
    let (π, φ) = abstract_next_path f S_0 true in
    if π = NoMorePath then break
    let (π', ·) = concretize π φ in
    if π' ≠ NoMorePath then
      update coverage information using π'

concretize π φ =
  Input: path π, path constraint φ
  Output: concretized path π' and path constraint φ' for π'
   or NoMorePath if (π, φ) cannot be concretized
  match π with
    case ϵ:
      return (ϵ, φ)
    case (S, ℓ : op) :: rest:
      match op with
        case m := g(x_1, ..., x_n):
          let S' = copy_args(S, op) in
          let (π', φ') = next_path g S' φ in
          if π' = NoMorePath then return (NoMorePath, ·)
          else
            let ψ = copy_ret m S φ' in
            if φ ∧ ψ is unsatisfiable then
              return (concretize π φ)
            else
              let (π'', φ'') =
                concretize rest (φ ∧ ψ) in
              if π'' = NoMorePath then
                return (concretize π φ)
              else return ((ℓ, op) :: (append π' π''), φ'' )
        otherwise :
          let (π', ·) = concretize rest φ in
          if π' = NoMorePath then return NoMorePath
          else return ((ℓ : op)::π', ·)

next_path g S φ =
  Input: Function g, symbolic store S, path constraint φ
  Output: Either NoMorePath or
   a concrete path π and a path constraint φ' for π
  let (π, φ) = abstract_next_path g S φ in
  match π with
    case NoMorePath: return (NoMorePath, ·)
    otherwise : return concretize π φ
```

Figure 5. The algorithms LATEST, *concretize*, and *next_path* for Lazy Expansion

symbolic store $\mathcal{S}_0$, and an initial path constraint `true`). The abstract path is concretized using a function *concretize* (described below). If the path can be concretized, then the concrete path is used to update coverage information (e.g., the branches visited). However, if the path cannot be concretized (*concretize* returns NoMorePath), the loop runs again to generate the next abstract path in f. The loop continues until there are no new abstract paths (i.e., *abstract_next_path* returns NoMorePath).

Given an abstract (intra-procedural) path through f, the *concretize* function is used to find a concretely realizable execution of the program by expanding out the called functions (and by recursively finding and concretizing abstract paths in the transitively called functions.) The input to *concretize* is an abstract path π that is annotated with the symbolic store at each point, and the path constraint ϕ of running π concolically. The function outputs a concrete (inter-procedural) execution whose projection on f is the path π together with the path constraint for the entire path, or returns NoMorePath indicating that the current path cannot be concretely realized.

The concretize function walks over the annotated path π and tries to justify each step by a concrete execution. An empty path is trivially justified. Otherwise, it looks at the first annotated operation $(\mathcal{S}, \ell : \mathsf{op})$ of the path, where ℓ is the program counter, $\mathcal{S}$ is the symbolic store used by the concolic execution at ℓ along this path, and op is the program operation. The concretize function considers the following two cases of op.

If op is a function call $m := g(x_1, \ldots, x_n)$, then *concretize* has to expand the current call to g to find a concretely realizable path in g that can be stitched to the path in f. In order to do this, the current symbolic store $\mathcal{S}'$, where symbolic expressions for the actual arguments to g in $\mathcal{S}$ are copied to the formal arguments of g (performed by function *copy_args*) is computed. A concretely realizable path in g, starting from $\mathcal{S}'$ and also satisfying the path constraint ϕ, is obtained by the function *next_path* as follows. *next_path* finds a concrete path (if one exists) by recursively invoking the abstract concolic exploration function *abstract_next_path* on g, $\mathcal{S}'$, and ϕ, and then concretizing this abstract path using *concretize*. If there are no more concrete paths of g starting from $\mathcal{S}'$ and satisfying ϕ, then *concretize* returns NoMorePath. However, if the current path through g is such that it cannot be stitched together (i.e., there is no way to find inputs that simultaneously satisfy the constraint ϕ as well as the path constraint on the return value of g that constrains the value of m), then the next path in g is searched. If, on the other hand, one can find a concrete execution through g that can extend the current path, concretize recursively justifies the rest of the trace, and returns a stitched concrete inter-procedural path.

In this way, *concretize* performs a backtracking search through concrete executions of the called functions in order to find an inter-procedural valid execution.

If on the other hand, op is some other program operation (an assignment, conditional, or a return), then *concretize* recursively concretizes the rest of the path, and returns the inter-procedural path obtained by prepending the current operation $(\ell : \mathsf{op})$ to the recursively constructed path. If the rest of the path cannot be concretized, then *concretize* returns NoMorePath in the recursive call, and this is returned by the current call as well.

The algorithm uses the helper functions *copy_args* and *copy_ret* that perform symbolic copying of actual parameters to formal parameters and symbolic copy back of the return parameter to the caller, respectively. The descriptions of these functions are omitted. In particular, the function *copy_ret* is used to add a constraint $\alpha_m = \alpha_{\texttt{ret}}$ to the path constraint, where α_m was the free input corresponding to the return value from g assumed by *abstract_next_path*. This new constraint ensures that the return value is now constrained by the constraints imposed by the path through g.

The soundness and the completeness property of the LATEST algorithm is given with respect to the CUTE search algorithm.

THEOREM 1. **[***Relative Soundness and Completeness] Given a program f having a finite computation tree and given any statement* **s** *in f, CUTE executes* **s** *if and only if* LATEST *executes* **s***.*

4.2 Implementation of Lazy Expansion

In the presence of side effects, the algorithm in the previous section can be unsound. This is because the updated path constraints may not reflect concrete updates to global state by the called function, as the following example illustrates.

EXAMPLE 1: The code in Figure 6 demonstrates that simply conjoining additional constraints on the path may lead to unsound results. In the example, the label `L` is unreachable, since the function `foo` sets the `global` bit to 1, thus invalidating the conditional guarding `L`. The first phase of concolic execution generates the path

```
int global;
int foo(int u) {
  global = 1;
  if (u == 0) return 0; else return 1;
}

int main() {
  global = 0;
  y = input();
  x = foo(y);
  if (global == 0 && x == 0) {
L: ;
  }
}
```

Figure 6. Unsoundness of the algorithm from Figure 5 in the presence of side effects

constraint $\alpha_x = 0$ and the store maps the address of the lvalue `x` to α_x. When the function call `foo` is expanded, we get the additional constraint $\texttt{ret}(\texttt{foo}) = 0$. There are no other symbolic constraints (from the store or the path) since `foo` is purely concrete. This leads us to mistakenly conclude that the path to `L` can be executed, and that the path taking the `else` branch cannot be executed. □

In order to provide sound results in the presence of side effects, we have to ensure that the implementation of LATEST is oblivious to side effects in function calls. We ensure this by modifying the algorithm so that it is stateless. Intuitively, instead of working with explicit symbolic stores and path constraints, we work with the bitvector of conditionals that are executed along a certain path. Recall that this bitvector is constructed in the $abstract\_next\_path$ function (variable $path$). We informally describe the main changes in the algorithm, which is implemented in our tool.

First, we modify the $abstract\_next\_path$ function to take as inputs a function f, and two bit sequences b_1 and b_2. The bit sequence b_1 denotes the sequence of conditionals that have been executed to reach the call to function f. The bit sequence b_2 gives the sequence of conditionals executed along the entire abstract execution path. We maintain the invariant that b_1 is a prefix of b_2. Given b_1 and b_2, we can reconstruct the symbolic store $\mathcal{S}_0$ and the path constraint ϕ, by performing concolic execution along the path determined by b_2.

Next, we change $concretize$ to make it side-effect oblivious and stateless. Similar to $abstract\_next\_path$, instead of propagating path constraints, we shall propagate bitvectors. The function $concretize$ will take a bitvector representing the conditionals executed along an abstract execution path and attempt to find a concrete execution through the program. When the next statement along the abstract path is a function call, $concretize$ executes the bitvector sequence concolically to generate a symbolic store at the function call and starts finding a concrete path in the function. However, when the function returns, we do not simply conjoin its path constraint to the path constraint of the original function. We try to concolically execute the partially expanded path to find inputs that can force the execution down the partially expanded path. Precisely, suppose that $concretize$ is called with a bitvector sequence $\langle b_0 : b_1 \rangle$, where the bitvector b_0 represents a partially expanded path up to the function call to g, and b_1 is the rest of the abstract path after the return from g. Then, $concretize$ will perform concolic execution to generate the symbolic store after b_0, then recursively expand the function call to g. When the expansion of g hits the return statement of g, instead of returning from the recursive call, it will check if the current store (at the return point of g) can be extended to execute b_1. This process is similar to the counterexample refinement algorithm using concolic execution from [11]. If the path can be extended, $concretize$ will return the partially expanded path $\langle b_0' : b_1 \rangle$, where the concrete path through g (and its transitively called functions) have been appended to b_0 to get b_0', and then find a concrete realization of the rest of the path b_1. Since the concolic execution explicitly executes the program along the path, we ensure that the algorithm is oblivious to side effects.

```
// locate index of first character c in s
int locate(char *s, int c) {
  int i = 0;
  while (s[i] != c) {
    if (s[i] == 0) return -1;
    i++;
  }
  return i;
}

void top(char *input) {
  int z;
  z = locate(input, 'a');
  if (z == -1) return -1;
  if (input[z+1] != ':') return 1;
  return 0;
}
```

Figure 7. Comparison with summarization (from [9])

4.3 Comparison with Summarization

For ease of exposition, we have presented the algorithm without memoization. However, we can apply summarization at function boundaries (similar to [22, 9]) in $concretize$, that first checks if a concrete path in the current function with the current symbolic constraints has been seen before (and if so, returns it), and only performs the exploration if a summary has not been seen before. The result is an algorithm that is similar to [9], but which can explore many fewer paths in testing the function-under-test. Unlike [9], where all possible execution summaries for a called function g are constructed as soon as g is called, we only construct enough summaries in the called functions to explore all the paths in the top level function. Since we work under a stronger, but realistic, assumption that all the library calls have already been (separately) tested, we can afford this optimization.

Figure 7 shows an example from [9]. For a string `s` of length less than or equal to n, the function `locate` has $2n$ distinct executions for any non-zero character `c` (and at most n if `c` is zero). There are n possible return values: -1 if `c` does not occur in `s`, and any index i in 0 to $n-1$ if `c` occurs in the ith place in `s`.

The function `top` calls `locate`, and itself has three different paths: either `locate` returns a -1 (i.e., 'a' is not present in the input), or 'a' is present in the input and is either followed by a ':' or not.

Usual concolic execution without lazy expansion or summarization searches $3n - 1$ paths in the code. Of these, n paths terminate after the then branch on line 1, n paths terminate after the then branch on line 2, and $n - 1$ terminate on line 3. The summary-based concolic execution of [9] computes a summary of `locate` by executing the $2n$ executions in `locate`, creating a summary with $2n$ terms, and uses this summary subsequently when analyzing the three paths in `top`. In all, it performs $2n + 3$ iterations: $2n$ for `locate` and 3 for `top`.

In comparison, lazy expansion searches 10 paths (which is independent of n) in the code. This is because LATEST does not attempt to explore all possible paths through the entire program, but to selectively find paths through `locate` that ensure full coverage of the function under test `top`. The first abstract path in `top` returns 1. This path gets concretized in 2 more iterations. The second ab-

Tool Name	Program Name	Running Time	# of Iterations	% Branch Coverage
CUTE	program1.c	30s	676	100.00
LATEST	program1.c	2s	59	100.00
CUTE	program2.c	21s	651	100.00
LATEST	program2.c	1.5s	63	100.00
CUTE	program3.c	20s	841	100.00
LATEST	program3.c	1.5s	68	100.00

Table 1. Testing 3 Simple Programs

Tool Name	# of Ops	Running Time	# of Iterations	# of valid Inputs	% Branch Coverage
CUTE	1	20m26s	19435	6	97.62
LATEST	1	3s	206	6	97.62
CUTE	2	$>$ 26hr	$> 10^6$	0	97.62
LATEST	2	47s	1671	26	97.62
CUTE	3	-	-	-	-
LATEST	3	6m8s	10000	104	97.62
CUTE	4	-	-	-	-
LATEST	4	431m11s	485256	3126	97.62

Table 2. Testing A Simple Calculator

stract path in `top` returns 0. This abstract path gets concretized in 4 iterations. The third abstract path returns -1 and this path gets concretized in 1 iteration (because the elements of `input` array gets initialized randomly and it is very unlikely that an element in this array is `'a'` or `':'`.)

Notice though that summarization and lazy expansion are *complementary* techniques. In the example above, every time we expand function `locate`, we can construct a path summary, and in subsequent expansions, we can first check if one of the already-constructed summaries provide a witness path.

4.4 Avoiding Abstraction of Input Independent Functions

In the abstract concolic exploration algorithm (Section 3), we abstract all functions that we encounter along an execution path. Observe that a number of such function's behavior may not depend on the program input. Such function calls will have a single path through them. Therefore, abstracting their return values with inputs will give rise to infeasible redundant abstract execution paths. In the implementation of LATEST, we dynamically identify if the input (or any memory read) by a function call has a mapping in the symbolic state. If not, then we execute the function without abstracting it. Otherwise, we abstract the function. This optimization prunes a lot of infeasible abstract execution paths.

5. Implementation and Evaluation

We have implemented LATEST on top of CUTE, a concolic unit testing engine for C [23]. Currently, LATEST requires us to identify the functions that we want to abstract. For example, in the parser case study we identify the `lex`, the `strcmp`, and the `regex` functions as abstract. LATEST has a switch: if the switch is off, then LATEST abstracts no function and falls back to concolic testing; if the switch is on, then LATEST invokes the lazy algorithm. The front-end of LATEST that instruments C code is written on top of CIL [18] and back-end that performs the actual symbolic execution is written in SmartEiffel. In this section, we report the results of our experiments with several programs, including simple calculator implementation and a parser of a simple imperative language called PL/0.

For each program, we describe the experimental setup and the results of comparing LATEST to concolic testing. We conducted the experiments on a Core 2 Duo Linux Desktop with 1 GB RAM.

5.1 Miscellaneous Small C Programs

We considered three small C programs: `program1.c` and `program2.c` are similar to the code in Figure 7, except that we make 2 calls to the `locate` function in `program1.c` on different input buffers and we make 2 calls to the `locate` function in `program2.c` on the same input buffer. `program3.c` is a variant of the code in Figure 1. Table 1 gives the results of running both CUTE and LATEST on these programs. In all cases, we got complete branch coverage. The table shows that number of iterations and the running time using LATEST is significantly smaller than that in CUTE.

5.2 A Simple Calculator

We considered a C implementation of a simple calculator that can perform basic operations such addition, subtraction, multiplication, and division. The implementation has around 120 lines of C code. Like any other calculator, this calculator takes numbers and operators as inputs. When we restrict the number of operations of the calculator along an execution path to one, the number of feasible execution paths of the calculator becomes less than 20000. This enables CUTE to explore all the paths in the calculator in a reasonable amount of time. We picked this example because we can run both LATEST and CUTE to completion and get an accurate comparison of the two techniques.

Table 2 shows the result of running both CUTE and LATEST on the simple calculator. The second column in the table gives the number of calculator operations that we performed along an execution path. We say an input is *valid* if the calculator produces no syntax error on the input. CUTE failed to terminate with merely two operations along an execution path. Moreover, after running for 26 hours, CUTE produced no valid input sequence for the calculator. On the other hand, LATEST completed testing in 47 seconds after generating 26 valid inputs. With one operation along a path, we attained a branch coverage of 97.62% with both tools. This provides evidence to our main theorem in Section 4. LATEST also managed to terminate when we had 4 operations along a path. For this case, we estimated that the number feasible execution paths is more than 10^{10} and CUTE will run for several days before completion. This simple case study shows that LATEST can quickly generate test inputs for complex programs, where CUTE fails to terminate.

5.3 PL/0 Parser

We next report our experience with a recursive descent parser implementation of PL/0 written in C. PL/0 is a simplified version of the general-purpose programming language Pascal. The language usually serves as an example of how to construct a compiler. It was originally introduced in the book, Algorithms + Data Structures = Programs, by Niklaus Wirth in 1975. It features quite limited language constructs: basic arithmetic operations on integers and no control-flow constructs other than "if" and "while" blocks. While these limitations makes writing real applications in this language impractical, it helps the compiler remain compact and simple. We picked an implementation of this language because the implementation is written completely in C without any use of the compiler tools such as yacc or lex. This helped us to annotate the functions that can be abstracted easily.

Table 3 shows the result of running both CUTE and LATEST on the parser. The parser has infinite number of executions paths, because a program in PL/0 can have unbounded size. In order to make the test input generation experiment tractable, we bounded the size of a program in PL/0 to a finite number from the set $\{7, 8\}$. We started with 7 tokens, because there in no valid program in PL/0 with less than 7 tokens. The second column gives the number of

Tool Name	# of Tokens	Running Time	# of Iterations	# of valid Inputs	% Branch Coverage
CUTE	7	> 24hr	$> 10^6$	0	-
LATEST	7	56m13s	98485	99	74.26
CUTE	8	-	-	-	-
LATEST	8	4hr37m	448046	399	74.26

Table 3. Testing a Parser of PL/0

tokens that we considered in an input. We say an input is valid if the PL/0 parser can parse the input without syntax error. The table shows that CUTE never managed to generate valid inputs when the number of tokens was restricted to 7 or 8. We estimated that CUTE will take many days to generate all valid inputs with 7 tokens. LATEST generated valid inputs in reasonable amount of time even when the number of tokens was 8 (we estimated LATEST can also generate inputs with 9 or 10 tokens if we allow LATEST to run for a couple of days.) We obtained a branch coverage of 74.26% with LATEST. We found by manual code inspection that the branch coverage cannot be increased beyond 74.26% due to the presence of dead code. This case study shows that LATEST can be effective in generating significantly large number of test inputs for parsers.

6. Conclusion

In the recent years, there has been a renewed interest in automated testing techniques due to the increasing affordability of powerful program analysis, model-checking, and theorem proving techniques. Several recent tools for automated testing such as DART, CUTE, and EXE have taken the first step in adapting ideas from program analysis and model-checking. These tools have shown the potential to automatically test real-world programs having extensive pointer and data structure usage. Unfortunately, these new generation tools suffer from the notorious path explosion problem. This scenario is quite comparable with the state explosion problem that haunted the model checking community few years ago. The state explosion problem in model checking has been tackled partly by developing sophisticated techniques for abstraction, automated theorem proving, and reduction. Now we need to develop similar techniques in the domain of automated test generation.

In this paper, we develop a new technique to tackle the path explosion problem by adapting a well-known idea from static program analysis. In the context of dynamic test generation, we show that counterexample-guided refinement can be effective in quickly generating inputs for complex programs. Our experimental results validate this fact.

The results in this paper show that there is a great opportunity for improving systematic automated testing by bringing in ideas from program analysis and model checking. The challenge that remains is to figure out right ways of adapting various program analysis techniques in testing.

References

[1] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[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] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer, 2000.

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

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

[8] 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.

[9] P. Godefroid. Compositional dynamic test generation. In *POPL 07: Principles of Programming Languages*. ACM, 2007.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI 05: Programming Language Design and Implementation*, 2005.

[11] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. SYNERGY: a new algorithm for property checking. In *SIGSOFT FSE 06*, pages 117–127. ACM, 2006.

[12] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[13] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 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] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE 07: International Conference on Software Engineering*. ACM, 2007.

[17] J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, Lecture Notes in Computer Science 2304, pages 213–228. Springer, 2002.

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

[20] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, 2005.

[21] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE'07: International Conference on Software Engineering*, 2007.

[22] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

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

[24] M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[25] M. Soffa, A. Mathur, and N. Gupta. Generating test data for branch coverage. In *ASE '00: Automated software engineering*, page 219. IEEE Computer Society, 2000.

[26] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA 04*, pages 97–107. ACM, 2004.

[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.