CUTE: A Concolic Unit Testing Engine for C

Koushik Sen, Darko Marinov, Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign
{ksen,marinov,agha}@cs.uiuc.edu

ABSTRACT

In *unit testing*, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function. The entry function may contain pointer arguments, in which case the inputs to the unit are *memory graphs*. The paper addresses the problem of automating unit testing with memory graphs as inputs. The approach used builds on previous work combining *symbolic* and *concrete execution*, and more specifically, using such a combination to *generate test inputs* to explore all feasible execution paths. The current work develops a method to represent and track constraints that capture the behavior of a symbolic execution of a unit with memory graphs as inputs. Moreover, an efficient constraint solver is proposed to facilitate incremental generation of such test inputs. Finally, CUTE, a tool implementing the method is described together with the results of applying CUTE to real-world examples of C code.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability,Verification

Keywords: concolic testing, random testing, explicit path model-checking, data structure testing, unit testing, testing C programs.

1. INTRODUCTION

Unit testing is a method for modular testing of a programs' functional behavior. A program is decomposed into units, where each unit is a collection of functions, and the units are independently tested. Such testing requires specification of values for the inputs (or *test inputs*) to the unit. Manual specification of such values is labor intensive and cannot guarantee that all possible behaviors of the unit will be observed during the testing.

In order to improve the range of behaviors observed (or *test coverage*), several techniques have been proposed to automatically generate values for the inputs. One such technique is to *randomly* choose the values over the domain of potential inputs [4,8,10,21]. 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].

One approach which addresses the problem of redundant executions and increases test coverage is *symbolic execution* [1, 3, 9, 22, 23, 27, 28, 30]. 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*. Observe that the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to generate concrete values for inputs which would result in different paths being taken. The classic approach is to use depth first exploration of the paths by backtracking [14]. Unfortunately, for large or complex units, it is computationally intractable to precisely maintain and solve the constraints required for test generation.

To the best of our knowledge, Larson and Austin were the first to propose combining concrete and symbolic execution [16]. In their approach, the program is executed on some user-provided concrete input values. Symbolic path constraints are generated for the specific execution. These constraints are solved, if feasible, to see whether there are potential input values that would have led to a violation along the same execution path. This improves coverage while avoiding the computational cost associated with full-blown symbolic execution which exercises all possible execution paths.

Godefroid et al. proposed incrementally generating test inputs by combining concrete and symbolic execution [11]. In Godefroid et al.'s approach, during a concrete execution, a conjunction of symbolic constraints along the path of the execution is generated. These constraints are modified and then solved, if feasible, to generate further test inputs which would direct the program along alternative paths. Specifically, they systematically negate the conjuncts in the path constraint to provide a depth first exploration of all paths in the computation tree. If it is not feasible to solve the modified constraints, Godefroid et al. propose simply substituting random concrete values.

A challenge in applying Godefroid et al.'s approach is to provide methods which extract and solve the constraints generated by a program. This problem is particularly complex for programs which have dynamic data structures using

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

pointer operations. For example, pointers may have aliases. Because alias analysis may only be approximate in the presence of pointer arithmetic, using symbolic values to precisely track such pointers may result in constraints whose satisfaction is undecidable. This makes the generation of test inputs by solving such constraints infeasible. In this paper, we provide a method for representing and solving *approximate pointer constraints* to generate test inputs. Our method is thus applicable to a broad class of sequential programs.

The key idea of our method is to represent inputs for the unit under test using a *logical input map* that represents all inputs, including (finite) memory graphs, as a collection of scalar symbolic variables and then to build constraints on these inputs by symbolically executing the code under test.

We first instrument the code being tested by inserting function calls which perform symbolic execution. We then repeatedly run the instrumented code as follows. The logical input map $\mathcal{I}$ is used to generate concrete memory input graphs for the program and two symbolic states, one for pointer values and one for primitive values. The code is run concretely on the concrete input graph and symbolically on the symbolic states, collecting constraints (in terms of the symbolic variables in the symbolic state) that characterize the set of inputs that would (likely) take the same execution path as the current execution path. As in [11], one of the collected constraints is negated. The resulting constraint system is solved to obtain a new logical input map $\mathcal{I}'$ that is similar to $\mathcal{I}$ but (likely) leads the execution through a different path. We then set $\mathcal{I} = \mathcal{I}'$ and repeat the process. Since the goal of this testing approach is to explore feasible execution paths as much as possible, it can be seen as *Explicit Path Model-Checking.*

An important contribution of our work is separating pointer constraints from integer constraints and keeping the pointer constraints simple to make our *symbolic execution light-weight* and our constraint solving procedure *not only tractable but also efficient.* The pointer constraints are conceptually simplified using the logical input map to replace complex symbolic expressions involving pointers with simple symbolic pointer variables (while maintaining the precise pointer relations in the logical input map). For example, if `p` is an input pointer to a `struct` with a field `f`, then a constraint on `p->f` will be simplified to a constraint on f_0, where f_0 is the symbolic variable corresponding to the input value `p->f`. Although this simplification introduces some approximations that do not precisely capture all executions, it results in simple pointer constraints of the form $x = y$ or $x \neq y$, where x and y are either symbolic pointer variables or the constant `NULL`. These constraints can be efficiently solved, and the approximations seem to suffice in practice.

We implemented our method in a tool called CUTE (*C*oncolic *U*nit *T*esting *E*ngine, where *Concolic* stands for cooperative *Conc*rete and symb*olic* execution). CUTE is available at `http://osl.cs.uiuc.edu/~ksen/cute/`. CUTE implements a solver for both arithmetic and pointer constraints to incrementally generate test inputs. The solver exploits the domain of this particular problem to implement three novel optimizations which help to improve the testing time by several orders of magnitude. Our experimental results confirm that CUTE can efficiently explore paths in C code, achieving high branch coverage and detecting bugs. In particular, it exposed software bugs that result in assertion violations, segmentation faults, or infinite loops.

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int
f(int v) {
  return 2*v + 1;
}

int
testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          ERROR;
  return 0;
}
```

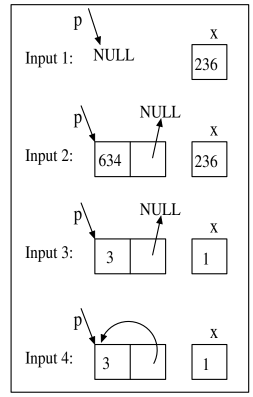

Figure 1: Example C code and inputs that CUTE generates for testing the function `testme`

This paper presents two case studies of testing code using CUTE. The first study involves the C code of the CUTE tool itself. The second case study found two previously unknown errors (a segmentation fault and an infinite loop) in SGLIB [25], a popular C data structure library used in a commercial tool. We reported the SGLIB errors to the SGLIB developers who fixed them in the next release.

2. EXAMPLE

We use a simple example to illustrate how CUTE performs testing. Consider the C function `testme` shown in Figure 1. This function has an error that can be reached given some specific values of the input. In a narrow sense, the input to `testme` consists of the values of the arguments `p` and `x`. However, `p` is a pointer, and thus the input includes the memory graph reachable from that pointer. In this example, the graph is a list of `cell` allocation units.

For the example function `testme`, CUTE first non-randomly generates `NULL` for `p` and randomly generates 236 for `x`, respectively. Figure 1 shows this input to `testme`. As a result, the first execution of `testme` takes the `then` branch of the first `if` statement and the `else` branch of the second `if`. Let p_0 and x_0 be the symbolic variables representing the values of `p` and `x`, respectively, at the beginning of the execution. CUTE collects the constraints from the predicates of the branches executed in this path: $x_0 > 0$ (for the `then` branch of the first `if`) and $p_0 = \texttt{NULL}$ (for the `else` branch of the second `if`). The predicate sequence $\langle x_0 > 0, p_0 = \texttt{NULL}\rangle$ is called a *path constraint.*

CUTE next solves the path constraint $\langle x_0 > 0, p_0 \neq \texttt{NULL}\rangle$, obtained by negating the last predicate, to drive the next execution along an alternative path. The solution that CUTE proposes is $\{p_0 \mapsto \text{non-}\texttt{NULL}, x_0 \mapsto 236\}$, which requires that CUTE make `p` point to an allocated `cell` that introduces two new components, `p->v` and `p->next`, to the reachable graph. Accordingly, CUTE randomly generates 634 for `p->v` and non-randomly generates `NULL` for `p->next`, respectively, for the next execution. In the second execution, `testme` takes the `then` branch of the first and the second `if` and the `else` branch of the third `if`. For this execution, CUTE generates the path constraint $\langle x_0 > 0, p_0 \neq \texttt{NULL}, 2 \cdot x_0 + 1 \neq v_0\rangle$, where p_0, v_0, n_0, and x_0 are the symbolic values of `p`, `p->v`, `p->next`, and `x`, respectively. Note that CUTE computes the expression

$2 \cdot x_0 + 1$ (corresponding to the execution of `f`) through an inter-procedural, dynamic tracing of symbolic expressions.

CUTE next solves the path constraint $\langle x_0 > 0, p_0 \neq \texttt{NULL}, 2 \cdot x_0 + 1 = v_0 \rangle$, obtained by negating the last predicate and generates Input 3 from Figure 1 for the next execution. Note that the specific value of x_0 has changed, but it remains in the same equivalence class with respect to the predicate where it appears, namely $x_0 > 0$. On Input 3, `testme` takes the `then` branch of the first three `if` statements and the `else` branch of the fourth `if`. CUTE generates the path constraint $\langle x_0 > 0, p_0 \neq \texttt{NULL}, 2 \cdot x_0 + 1 = v_0, p_0 \neq n_0 \rangle$. This path constraint includes dynamically obtained constraints on pointers. CUTE handles constraints on pointers but requires no static alias analysis. To drive the program along an alternative path in the next execution, CUTE solves the constraints $\langle x_0 > 0, p_0 \neq \texttt{NULL}, 2 \cdot x_0 + 1 = v_0, p_0 = n_0 \rangle$ and generates Input 4 from Figure 1. On this input, the fourth execution of `testme` reveals the error in the code.

3. CUTE

We first define the input logical input map that CUTE uses to represent inputs. We also introduce program units of a simple C-like language (*cf.* [19]). We present how CUTE instruments programs and performs concolic execution. We then describe how CUTE solves the constraints after every execution. We next present how CUTE handles complex data structures. We finally discuss the approximations that CUTE uses for pointer constraints.

To explore execution paths, CUTE first instruments the code under test. CUTE then builds a *logical input map* $\mathcal{I}$ for the code under test. Such a logical input map can represent a memory graph in a symbolic way. CUTE then repeatedly runs the instrumented code as follows:

1. It uses the logical input map $\mathcal{I}$ to generate a concrete input memory graph for the program and two symbolic states, one for pointer values and another for primitive values.
2. It runs the code on the concrete input graph, collecting constraints (in terms of the symbolic values in the symbolic state) that characterize the set of inputs that would take the same execution path as the current execution path.
3. It negates one of the collected constraints and solves the resulting constraint system to obtain a new logical input map $\mathcal{I}'$ that is similar to $\mathcal{I}$ but (likely) leads the execution through a different path. It then sets $\mathcal{I} = \mathcal{I}'$ and repeats the process.

Conceptually, CUTE executes the code under test both concretely and symbolically at the same time. The actual CUTE implementation first instruments the source code under test, adding functions that perform the symbolic execution. CUTE then repeatedly executes the instrumented code only concretely.

3.1 Logical Input Map

CUTE keeps track of input memory graphs as a logical input map $\mathcal{I}$ that maps logical addresses to values that are either logical addresses or primitive values. This map symbolically represents the input memory graph at the beginning of an execution. The reason that CUTE introduces logical addresses is that actual concrete addresses of dynamically allocated cells may change in different executions. Also, the concrete addresses themselves are not necessary to represent memory graphs; it suffices to know how the cells are connected. Finally, CUTE attempts to make consecutive inputs similar, and this can be done with logical addresses. If CUTE used the actual physical addresses, it would depend on `malloc` and `free` (to return the same addresses) and more importantly, it would need to handle destructive updates of the input by the code under test: after CUTE generates one input, the code changes it, and CUTE would need to know what changed to reconstruct the next input.

Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{V}$ be the set of all primitive values. Then, $\mathcal{I} \colon \mathbb{N} \to \mathbb{N} \cup \mathbb{V}$. The values in the domain and the range of $\mathcal{I}$ belonging to the set $\mathbb{N}$ represents the logical addresses. We also assume that each logical address $l \in \mathbb{N}$ has a type associated with it. A type can be `T *` (a pointer of type `T`) (where `T` can be primitive type or struct type) or $\texttt{T}_p$ (a primitive type). The function *typeOf*(l) returns this type. Let the function *sizeOf*$(\texttt{T})$ returns the number of memory cells that an object of type `T` uses. If *typeOf*(l) is `T *` and $\mathcal{I}(l) \neq$`NULL`, then the sequence $\mathcal{I}(v), \ldots, \mathcal{I}(v+n-1)$ stores the value of the object pointed by the logical address l (each element in the sequence represents the content of each cell of the object in order), where $v = \mathcal{I}(l)$ and $n =$*sizeOf*$(\texttt{T})$. This representation of a logical input map essentially gives a simple way to serialize a memory graph.

We illustrate logical inputs on an example. Recall the example Input 3 from Figure 1. CUTE represents this input with the following logical input: $\langle 3, 1, 3, 0 \rangle$, where logical addresses range from 1 to 4. The first value 3 corresponds to the value of `p`: it points to the location with logical address 3. The second value 1 corresponds to `x`. The third value corresponds to `p->v` and the fourth to `p->next` (0 represents `NULL`). This logical input encodes a set of concrete inputs that have the same underlying graph but reside at different concrete addresses. Similarly, the logical input map for Input 4 from Figure 1 is $\langle 3, 1, 3, 3 \rangle$.

3.2 Units and Program Model

A unit under test can have several functions. CUTE requires the user to select one of them as the *entry* function for which CUTE generates inputs. This function in turn can call other functions in the unit as well as functions that are not in the unit (e.g., library functions). The entry function takes as input a memory graph, a set of all memory locations reachable from the input pointers. We assume that the unit operates only on this input, i.e., the unit has no external functions (that would, for example, simulate an interactive input from the user or file reading). However, a program can allocate additional memory, and the execution then operates on some locations that were not reachable in the initial state. Given an entry function, CUTE generates a `main` function that first initializes all the arguments of the function by calling the primitive function *input*() (described next) and then calls the entry function with these arguments. The unit along with the `main` function forms a closed program that CUTE instruments and tests.

We describe how CUTE works for a simple C-like language shown in Figure 2. `START` represents the first statement of a program under test. Each statement has an optional label. The program can get input using the expression *input()*. For simplicity of description, we assume that a program gets all

$$
\begin{array}{lll}
P ::= & \mathit{Stmt}^* & \mathit{Stmt} ::= [l\!:]\ S \\
S ::= & \multicolumn{2}{l}{\mathit{lhs} \leftarrow e \mid \texttt{if}\ p\ \texttt{goto}\ l' \mid \texttt{START} \mid \texttt{HALT} \mid \texttt{ERROR}} \\
\mathit{lhs} ::= & v \mid *v & \\
e ::= & \multicolumn{2}{l}{v \mid \&v \mid *v \mid c \mid v\ op\ v \mid \mathit{input}()} \\
& \multicolumn{2}{l}{\text{where } op \in \{+,-,/,*,\%,\ldots\},} \\
& \multicolumn{2}{l}{v \text{ is a variable, } c \text{ is a constant}} \\
p ::= & \multicolumn{2}{l}{v = v \mid v \neq v \mid v < v \mid v \leq v \mid v \geq v \mid v > v}
\end{array}
$$

Figure 2: Syntax of a simple C-like language

Before Instrumentation	After Instrumentation
// program start START	*global vars* $\mathcal{A} = \mathcal{P} = path\_c = M = [\,]$; *global vars* $i = inputNumber = 0$; START
// inputs $v \leftarrow input()$;	$inputNumber = inputNumber + 1$; $initInput(\&v, inputNumber)$;
// inputs $*v \leftarrow input()$;	$inputNumber = inputNumber + 1$; $initInput(v, inputNumber)$;
// assignment $v \leftarrow e$;	$execute\_symbolic(\&v, \text{“}e\text{”})$; $v \leftarrow e$;
// assignment $*v \leftarrow e$;	$execute\_symbolic(v, \text{“}e\text{”})$; $*v \leftarrow e$;
// conditional if (p) goto l	$evaluate\_predicate(\text{“}p\text{”}, p)$; if ($p$) goto l
// normal termination HALT	$solve\_constraint()$; HALT;
// program error ERROR	**print** “Found Error” ERROR;

Figure 3: Code that CUTE's instrumentation adds

the inputs at the beginning of an execution and the number of inputs is fixed. CUTE uses the CIL framework [19] to convert more complex statements (with no function calls) into this simplified form by introducing temporary variables. For example, CIL converts `**v = 3` into `t1 = *v; *t1 = 3` and `p[i] = q[j]` into `t1 = q+j; t2 = p+i; *t2 = *t1`. Details of handling of function calls using a symbolic stack are discussed in [24].

The C expression $\&v$ denotes the address of the variable v, and $*v$ denotes the value of the address stored in v. In concrete state, each address stores a value that either is *primitive* or represents another memory address (*pointer*).

3.3 Instrumentation

To test a program P, CUTE tries to explore all execution paths of P. To explore all paths, CUTE first instruments the program under test. Then, it repeatedly runs the instrumented program P as follows:

```
// input: P is the instrumented program to test
//        depth is the depth of bounded DFS
run_CUTE(P,depth)
  I = [ ]; h = (number of arguments in P) + 1;
  completed=false; branch_hist=[ ];
  while not completed
    execute P
```

Before starting the execution loop, CUTE initializes the logical input map $\mathcal{I}$ to an empty map and the variable h representing the next available logical address to the number of arguments to the instrumented program plus one. (CUTE gives a logical address to each argument at the very beginning.) The integer variable *depth* specifies the depth in the bounded DFS described in Section 3.4.

Figure 3 shows the code that CUTE adds during instrumentation. The expressions enclosed in double quotes (“e”) represent syntactic objects. Due to space constraint, we describe the instrumentation for function calls in [24]. In the following section, we describe the various global variables and procedures that CUTE inserts.

3.4 Concolic Execution

Recall that a program instrumented by CUTE runs concretely and at the same time performs symbolic computation through the instrumented function calls. The symbolic execution follows the path taken by the concrete execution and replaces with the concrete value any symbolic expression that cannot be handled by our constraint solver.

An instrumented program maintains at the runtime two *symbolic states* $\mathcal{A}$ and $\mathcal{P}$, where $\mathcal{A}$ maps memory locations to *symbolic arithmetic expressions*, and $\mathcal{P}$ maps memory locations to *symbolic pointer expressions*. The symbolic arithmetic expressions in CUTE are linear, i.e. of the form $a_1x_1+\ldots+a_nx_n+c$, where $n \geq 1$, each x_i is a symbolic variable, each a_i is an integer constant, and c is an integer constant. Note that n must be greater than 0. Otherwise, the expression is a constant, and CUTE does not keep constant expressions in $\mathcal{A}$, because it keeps $\mathcal{A}$ small: if a symbolic expression is constant, its value can be obtained from the concrete state. The arithmetic constraints are of the form $a_1x_1 + \ldots + a_nx_n + c \bowtie 0$, where $\bowtie \in \{<,>,\leq,\geq,=,\neq\}$. The pointer expressions are simpler: each is of the form x_p, where x_p is a symbolic variable, or the constant NULL. The pointer constraints are of the form $x \cong y$ or $x \cong \texttt{NULL}$, where $\cong \in \{=,\neq\}$.

Given any map $\mathcal{M}$ (e.g., $\mathcal{A}$ or $\mathcal{P}$), we use $\mathcal{M}' = \mathcal{M}[m \mapsto v]$ to denote the map that is the same as $\mathcal{M}$ except that $\mathcal{M}'(m) = v$. We use $\mathcal{M}' = \mathcal{M} - m$ to denote the map that is the same as $\mathcal{M}$ except that $\mathcal{M}'(m)$ is undefined. We say $m \in domain(\mathcal{M})$ if $\mathcal{M}(m)$ is defined.

Input Initialization using Logical Input Map

Figure 4 shows the procedure $initInput(m, l)$ that uses the logical input map $\mathcal{I}$ to initialize the memory location m, to update the symbolic states $\mathcal{A}$ and $\mathcal{P}$, and to update the input map $\mathcal{I}$ with new mappings.

M maps logical addresses to physical addresses of memory cells already allocated in an execution, and $malloc(n)$ allocates n fresh cells for an object of size n and returns the addresses of these cells as a sequence. The global variable h keeps track of the next unused logical address available for a newly allocated object.

For a logical address l passed as an argument to *initInput*, $\mathcal{I}(l)$ can be undefined in two cases: (1) in the first execution when $\mathcal{I}$ is the empty map, and (2) when l is some logical address that got allocated in the process of initialization. If $\mathcal{I}(l)$ is undefined and if $typeOf(l)$ is not a pointer, then the content of the memory is initialized *randomly*; otherwise, if the $typeOf(l)$ is a pointer, then the contents of l and m are both initialized to NULL. Note that CUTE does not attempt to generate random pointer graphs but assigns all new pointers to NULL. If $typeOf(\mathcal{I}(l))$ is a pointer to `T` (i.e., `T *`) and $M(l)$ is defined, then we know that the object pointed by the logical address l is already allocated and we simply initialize the content of m by $M(l)$. Otherwise, we allocate sufficient physical memory for the object pointed by `*m` using *malloc* and initialize them recursively. In the

```
// input: m is the physical address to initialize
//          l is the corresponding logical address
// modifies h, I, A, P
initInput(m, l)
  if l ∉ domain(I)
    if (typeOf(*m) ==pointer to T) *m =NULL;
    else *m =random();
    I = I[l ↦ *m];
  else
    v' = v = I(l);
    if (typeOf(v) ==pointer to T)
      if (v ∈ domain(M))
        *m = M(v);
      else
        n = sizeOf(T);
        {m_1, ..., m_n} =malloc(n);
        if (v ==non-NULL)
          v' = h; h = h + n; // h is the next logical address
        *m = m_1; I = I[l ↦ v']; M = M[v ↦ m_1];
        for j = 1 to n
          input(m_j, h + j − 1);
    else
      *m = v; I = I[l ↦ v];
  // x_l is a symbolic variable for logical address l
  if (typeOf(m) ==pointer to T) P = P[m ↦ x_l];
  else A = A[m ↦ x_l];
```

Figure 4: Input initialization

process, we also allocate logical addresses by incrementing h if necessary.

Symbolic Execution

Figure 5 shows the pseudo-code for the symbolic manipulations done by the procedure *execute_symbolic* which is inserted by CUTE in the program under test during instrumentation. The procedure *execute_symbolic*(m, e) evaluates the expression e symbolically and maps it to the memory location m in the appropriate symbolic state.

Recall that CUTE replaces a symbolic expression that the CUTE's constraint solver cannot handle with the concrete value from the execution. Assume, for instance, that the solver can solve only linear constraints. In particular, when a symbolic expression becomes non-linear, as in the multiplication of two non-constant sub-expressions, CUTE simplifies the symbolic expression by replacing one of the sub-expressions by its current concrete value (see line L in Figure. 5). Similarly, if the statement is for instance $v'' \leftarrow v/v'$ (see line D in Figure. 5), and both v and v' are symbolic, CUTE removes the memory location $\&v''$ from both $\mathcal{A}$ and $\mathcal{P}$ to reflect the fact that the symbolic value for v'' is undefined.

Figure 6 shows the function *evaluate_predicate*(p, b) that symbolically evaluates p and updates *path_c*. In case of pointers, CUTE only considers predicates of the form $x = y$, $x \neq y$, $x =$`NULL`, and $x \neq$`NULL`, where x and y are symbolic pointer variables. We discuss this in Section 3.7. If a symbolic predicate expression is constant, then `true` or `false` is returned.

At the time symbolic evaluation of predicates in the procedure *evaluate_predicate*, symbolic predicate expressions from branching points are collected in the array *path_c*. At the end of the execution, *path_c*$[0 \ldots i-1]$, where i is the number of conditional statements of P that CUTE executes, contains all predicates whose *conjunction* holds for the execution path.

Note that in both the procedures *execute_symbolic* and

```
// inputs: m is a memory location
//           e is an expression to evaluate
// modifies A and P by symbolically executing *m ← e
execute_symbolic(m, e)
 if (i ≤depth)
  match e:
    case "v_1":
      m_1 = &v_1;
      if (m_1 ∈ domain(P))
        A = A − m; P = P[m ↦ P(m_1)]; // remove if A contains m
      else if (m_1 ∈ domain(A))
        A = A[m ↦ A(m_1)]; P = P − m;
      else P = P − m; A = A − m;
    case "v_1 ± v_2": // where ± ∈ {+, −}
      m_1 = &v_1; m_2 = &v_2;
      if (m_1 ∈ domain(A) and m_2 ∈ domain(A))
        v = "A(m_1) ± A(m_2)"; // symbolic addition or subtraction
      else if (m_1 ∈ domain(A))
        v = "A(m_1) ± v_2"; // symbolic addition or subtraction
      else if (m_2 ∈ domain(A))
        v = "v_1 ± A(m_2)"; // symbolic addition or subtraction
      else A = A − m; P = P − m; return;
      A = A[m ↦ v]; P = P − m;
    case "v_1 * v_2":
      m_1 = &v_1; m_2 = &v_2;
      if (m_1 ∈ domain(A) and m_2 ∈ domain(A))
L:      v = "v_1 * A(m_2)"; // replace one with concrete value
      else if (m_1 ∈ domain(A))
        v = "A(m_1) * v_2"; // symbolic multiplication
      else if (m_2 ∈ domain(A))
        v = "v_1 * A(m_2)"; // symbolic multiplication
      else A = A − m; P = P − m; return;
      A = A[m ↦ v]; P = P − m;
    case "*v_1":
      m_2 = v_1;
      if (m_2 ∈ domain(P)) A = A − m; P = P[m ↦ P(m_2)];
      else if (m_2 ∈ domain(A)) A = A[m ↦ A(m_2)]; P = P − m;
          else A = A − m; P = P − m;
    default:
D:    A = A − m; P = P − m;
```

Figure 5: Symbolic execution

evaluate_predicate, we skip symbolic execution if the number of predicates executed so far (recorded in the global variable i) becomes greater than the parameter *depth*, which gives the depth of bounded DFS described next.

Bounded Depth-First Search

To explore paths in the execution tree, CUTE implements a *(bounded) depth-first* strategy (bounded DFS). In the bounded DFS, each run (except the first) is executed with the help of a record of the conditional statements (which is the array *branch_hist*) executed in the previous run. The procedure *cmp_n_set_branch_hist* in figure 7 checks whether the current execution path matches the one predicted at the end of the previous execution and represented in the variable *branch_hist*. We observed in our experiments that the execution almost always follows a prediction of the outcome of a conditional. However, it could happen that a prediction is not fulfilled because CUTE approximates, when necessary, symbolic expressions with concrete values (as explained in Section 3.4), and the constraint solver could then produce a solution that changes the outcome of some earlier branch. (Note that even when there is an approximation, the solution does not necessary change the outcome.) If it ever happens that a prediction is not fulfilled, an exception is raised to restart *run_CUTE* with a fresh random input.

Bounded depth-first search proves useful when the length of execution paths are infinite or long enough to prevent exhaustively search the whole computation tree. Particularly,

```
// inputs: p is a predicate to evaluate
//         b is the concrete value of the predicate in S
// modifies path_c, i
evaluate_predicate(p, b)
 if (i ≤ depth)
   match p:
     case "v1 ⋈ v2": // where ⋈ ∈ {<, ≤, ≥, >}
       m1 = &v1; m2 = &v2;
       if (m1 ∈ domain(A) and m2 ∈ domain(A))
         c = "A(m1) − A(m2) ⋈ 0";
       else if (m1 ∈ domain(A))
         c = "A(m1) − v2 ⋈ 0";
       else if (m2 ∈ domain(A))
         c = "v1 − A(m2) ⋈ 0";
       else c = b;
     case "v1 ≅ v2": // where ≅ ∈ {=, ≠}
       m1 = &v1; m2 = &v2;
       if (m1 ∈ domain(P) and m2 ∈ domain(P))
         c = "P(m1) ≅ P(m2)";
       else if (m1 ∈ domain(P) and v2 ==NULL)
         c = "P(m1) ≅ NULL";
       else if (m2 ∈ domain(P) and v1 ==NULL)
         c = "P(m2) ≅ NULL";
       else if (m1 ∈ domain(A) and m2 ∈ domain(A))
         c = "A(m1) − A(m2) ≅ 0";
       else if (m1 ∈ domain(A)) c = "A(m1) − v2 ≅ 0";
       else if (m2 ∈ domain(A)) c = "v1 − A(m2) ≅ 0";
       else c = b;
   if (b) path_c[i] = c;
   else path_c[i] =neg(c);
 cmp_n_set_branch_hist(true);
 i = i + 1;
```

Figure 6: Symbolic evaluation of predicates

```
// modifies branch_hist
cmp_n_set_branch_hist(branch)
  if (i < |branch_hist|)
    if (branch_hist[i].branch≠branch)
      print "Prediction Failed";
      raise an exception; // restart run_CUTE
    else if (i == |branch_hist| − 1)
      branch_hist[i].done =true;
  else branch_hist[i].branch = branch;
    branch_hist[i].done = false;
```

Figure 7: Prediction checking

it important for generating finite sized data structures when using preconditions such as data structure invariants (see section 3.6. For example, if we use an invariant to generate sorted binary trees, then a non-bounded depth-first search would end up generating infinite number of trees whose every node has at most one left children and no right children.

3.5 Constraint Solving

We next present how CUTE solves path constraints. Given a path constraint C=*neg_last*(*path_c*$[0\dots j]$), CUTE checks if C is satisfiable, and if so, finds a satisfying solution $\mathcal{I}'$.

We have implemented a constraint solver for CUTE to optimize solving of the path constraints that arise in concolic execution. Our solver is built on top of `lp_solve` [17], a constraint solver for linear arithmetic constraints. Our solver provides three important optimizations for path constraints:
(OPT 1) Fast unsatisfiability check: The solver checks if the last constraint is syntactically the negation of any preceding constraint; if it is, the solver does not need to invoke the expensive semantic check. (Experimental results show that this optimization reduces the number of semantic checks by 60-95%.)
(OPT 2) Common sub-constraints elimination: The

```
// modifies branch_hist, I, completed
solve_constraint() =
  j = i − 1;
  while (j ≥ 0)
    if (branch_hist[j].done == false)
      branch_hist[j].branch= ¬branch_hist[j].branch;
      if (∃I' that satisfies neg_last(path_c[0 . . . j]))
        branch_hist=branch_hist[0 . . . j];
        I = I';
        return;
      else j = j − 1;
    else j = j − 1;
  if (j < 0) completed=true;
```

Figure 8: Constraint solving

solver identifies and eliminates common arithmetic sub-constraints before passing them to the `lp_solve`. (This simple optimization, along with the next one, is significant in practice as it can reduce the number of sub-constraints by 64% to 90%.)
(OPT 3) Incremental solving: The solver identifies dependency between sub-constraints and exploits it to solve the constraints faster and keep the solutions similar. We explain this optimization in detail.

Given a predicate p in C, we define *vars*(p) to be the set of all symbolic variables that appear in p. Given two predicates p and p' in C, we say that p and p' are *dependent* if one of the following conditions holds:

1. *vars*$(p) \cap$ *vars*$(p') \neq \emptyset$, or
2. there exists a predicate p'' in C such that p and p'' are dependent and p' and p'' are dependent.

Two predicates are independent if they are not dependent.

The following is an important observation about the path constraints C and C' from two consecutive concolic executions: C and C' differ in the small number of predicates (more precisely, only in the last predicate when there is no backtracking), and thus their respective solutions $\mathcal{I}$ and $\mathcal{I}'$ must agree on many mappings. Our solver exploits this observation to provide more efficient, incremental constraint solving. The solver collects all the predicates in C that are dependent on $\neg$*path_c*$[j]$. Let this set of predicates be D. Note that all predicates in D are either linear arithmetic predicates or pointer predicates, because no predicate in C contains both arithmetic symbolic variables and pointer symbolic variables. The solver then finds a solution $\mathcal{I}''$ for the conjunction of all predicates from D. The input for the next run is then $\mathcal{I}' = \mathcal{I}[\mathcal{I}'']$ which is the same as $\mathcal{I}$ except that for every l for which $\mathcal{I}''(l)$ is defined, $\mathcal{I}'(l) = \mathcal{I}''(l)$. In practice, we have found that the size of D is almost *one-eighth* the size of C on average.

If all predicates in D are linear arithmetic predicates, then CUTE uses *integer linear programming* to compute $\mathcal{I}''$. If all predicates in D are pointer predicates, then CUTE uses the following procedure to compute $\mathcal{I}''$.

Let us consider only pointer constraints, which are either equalities or disequalities. The solver first builds an equivalence graph based on (dis)equalities (similar to checking satisfiability in theory of equality [2]) and then based on this graph, assigns values to pointers. The values assigned to the pointers can be a logical address in the domain of $\mathcal{I}$, the constant `non-NULL` (a special constant), or the constant `NULL` (represented by 0). The solver views `NULL` as a symbolic variable. Thus, all predicates in D are of the form

```
// inputs: p is a symbolic pointer predicate
//         I is the previous solution
// returns: a new solution I''
solve_pointer(p, I)
  match p:
    case "x ≠NULL": I'' = {y ↦non-NULL| y ∈ [x]=};
    case "x =NULL": I'' = {y ↦NULL| y ∈ [x]=};
    case "x = y": I'' = {z ↦ v | z ∈ [y]= and I(x) = v};
    case "x ≠ y": I'' = {z ↦ non-NULL| z ∈ [y]=};
  return I'';
```

Figure 9: Assigning values to pointers

$x = y$ or $x \neq y$, where x and y are symbolic variables. Let D' be the subset of D that does not contain the predicate $\neg path\_c[j]$. The solver first checks if $\neg path\_c[j]$ is consistent with the predicates in D. For this, the solver constructs an undirected graph whose nodes are the equivalence classes (with respect to the relation $=$) of all symbolic variables that appear in D'. We use $[x]_=$ to denote the equivalence class of the symbolic variable x. Given two nodes denoted by the equivalence classes $[x]_=$ and $[y]_=$, the solver adds an edge between $[x]_=$ and $[y]_=$ iff there exists symbolic variables u and v such that $u \neq v$ exists in D' and $u \in [x]_=$ and $v \in [y]_=$. Given the graph, the solver finds that $\neg path\_c[j]$ is satisfiable if $\neg path\_c[j]$ is of the form $x = y$ and there is no edge between $[x]_=$ and $[y]_=$ in the graph; otherwise, if $\neg path\_c[j]$ is of the form $x \neq y$, then $\neg path\_c[j]$ is satisfiable if $[x]_=$ and $[y]_=$ are not the same equivalence class. If $\neg path\_c[j]$ is satisfiable, the solver computes $\mathcal{I}''$ using the procedure $solve\_pointer(\neg path\_c[j], \mathcal{I})$ shown in Figure 9.

Note that after solving the pointer constraints, we either add (by assigning a pointer to `non-NULL`) or remove a node (by assigning a pointer `NULL`) from the current input graph, or alias or non-alias two existing pointers. This keeps the consecutive solutions similar. Keeping consecutive solutions for pointers similar is important because of the logical input map: if inputs were very different, CUTE would need to rebuild parts of the logical input map.

3.6 Data Structure Testing

We next consider testing of functions that take data structures as inputs. More precisely, a function has some pointer arguments, and the memory graph reachable from the pointers forms a data structure. For instance, consider testing of a function that takes a list and removes an element from it. We cannot simply test such function in isolation [5,27,30]—say generating random memory graphs as inputs—because the function requires the input memory graph to satisfy the data structure *invariant*.[1] If an input is invalid (i.e., violates the invariant), the function provides no guarantees and may even result in an error. For instance, a function that expects an acyclic list may loop infinitely given a cyclic list, whereas a function that expects a cyclic list may dereference `NULL` given an acyclic list. We want to test such functions with valid inputs only. There are two main approaches to obtaining valid inputs: (1) generating inputs with call sequences [27, 30] and (2) solving data structure invariants [5, 27]. CUTE supports both approaches.

[1]The functions may have additional preconditions, but we omit them for brevity of discussion; for more details, see [5].

Generating Inputs with Call Sequences:

One approach to generating data structures is to use sequences of function calls. Each data structure implements functions for several basic operations such as creating an empty structure, adding an element to the structure, removing an element from the structure, and checking if an element is in the structure. A sequence of these operations can be used to generate an instance of data structure, e.g., we can create an empty list and add several elements to it. This approach has two requirements [27]: (1) all functions must be available (and thus we cannot test each function in isolation), and (2) all functions must be used in generation: for complex data structures, e.g., red-black trees, there are memory graphs that cannot be constructed through additions only but require removals [27, 30].

Solving Data Structure Invariants:

Another approach to generating data structures is to use the functions that check invariants. Good programming practice suggests that data structures provide such functions. For example, SGLIB [25] (see Section 4.2) is a popular C library for generic data structures that provides such functions. We call these functions `repOk` [5]. (SGLIB calls them `check_consistency`.) As an illustration, SGLIB implements operations on doubly linked lists and provides a `repOk` function that checks if a memory graph is a valid doubly linked list; each `repOk` function returns `true` or `false` to indicate the validity of the input graph.

The main idea of using `repOk` functions for testing is to *solve* `repOk` functions, i.e., generate only the input memory graphs for which `repOk` returns `true` [5, 27]. This approach allows modular testing of functions that implement data structure operations (i.e., does not require that all operations be available): all we need for a function under test is a corresponding `repOk` function. Previous techniques for solving `repOk` functions include a search that uses purely concrete execution [5] and a search that uses symbolic execution for primitive data but concrete values for pointers [27]. CUTE, in contrast, uses symbolic execution for both primitive data and pointers.

The constraints that CUTE builds and solves for pointers allow it to solve `repOk` functions asymptotically faster than the fastest previous techniques [5, 27]. Consider, for example, the following check from the invariant for doubly linked list: for each node `n`, `n.next.prev == n`. Assume that the solver is building a doubly linked list with N nodes reachable along the `next` pointers. Assume also that the solver needs to set the values for the `prev` pointers. Executing the check once, CUTE finds the exact value for each `prev` pointer and thus takes $O(N)$ steps to find the values for all N `prev` pointers. In contrast, the previous techniques [5, 27] take $O(N^2)$ steps as they search for the value for each pointer, trying first the value `NULL`, then a pointer to the head of the list, then a pointer to the second element and so on.

3.7 Approximations for Scalable Symbolic Execution

CUTE uses simple symbolic expressions for pointers and builds only (dis)equality constraints for pointers. We believe that these constraints, which approximate the exact path condition, are a good trade-off. To exactly track the pointer constraints, it would be necessary to use the theory of arrays/memory with updates and selections [18]. However, it would make the symbolic execution more expensive and could result in constraints whose solution is intractable. Therefore, CUTE does not use the theory of arrays but handles arrays by concretely instantiating them and making each element of the array a scalar symbolic variable.

It is important to note that, although CUTE uses simple pointer constraints, it still keeps a precise relationship between pointers: the logical input map (through types), maintains a relationship between pointers to structs and their fields and between pointers to arrays and their elements. For example, from the logical input map $\langle 3, 1, 3, 0\rangle$ for Input 3 from Figure 1, CUTE knows that `p->next` is at the (logical) address 4 because `p` has value 3, and the field `next` is at the offset 1 in the `struct cell`. Indeed, the logical input map allows CUTE to use only simple scalar symbolic variables to represent the memory and still obtain fairly precise constraints.

Finally, we show that CUTE does not keep the exact pointer constraints. Consider for example the code snippet `*p=0; *q=1; if (*p == 1) ERROR` (and assume that `p` and `q` are not `NULL`). CUTE cannot generate the constraint `p==q` that would enable the program to take the "then" branch. This is because the program contains no conditional that can generate the constraint. Analogously, for the code snippet `a[i]=0; a[j]=1; if (a[i]==0) ERROR`, CUTE cannot generate `i==j`.

4. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We have implemented the main parts of CUTE in C. To instrument code under test, we use CIL [19], a framework for parsing and transforming C programs. To solve arithmetic inequalities, the constraint solver of CUTE uses `lp_solve` [17], a library for integer linear programming. Further details about the implementation can be found in [24].

We illustrate two case studies that show how CUTE can detect errors. In the second case study, we also present results that show how CUTE achieves branch coverage of the code under test. We performed all experiments on a Linux machine with a dual 1.7 GHz Intel Xeon processor.

4.1 Data Structures of CUTE

We applied CUTE to test its own data structures. CUTE uses a number of non-standard data structures at runtime, such as `cu_linear` to represent linear expressions, `cu_pointer` to represent pointer expressions, `cu_depend` to represent dependency graphs for path constraints etc. Our goal in this case study was to detect memory leaks in addition to standard errors such as segmentation faults, assertion violation etc. To that end, we used CUTE in conjunction with valgrind [26]. We discovered a few memory leaks and a couple of segmentation faults that did not show up in other uses of CUTE. This case study is interesting in that we applied CUTE to partly unit test itself and discovered bugs. We briefly describe our experience with testing the `cu_linear` data structure.

We tested the `cu_linear` module of CUTE in the depth-first search mode of CUTE along with valgrind. In 537 iterations, CUTE found a memory leak. The following is a snippet of the function `cu_linear_add` relevant for the memory leak:

```
cu_linear *
cu_linear_add(cu_linear *c1, cu_linear *c2, int add) {
  int i, j, k, flag;
  cu_linear* ret=(cu_linear*)malloc(sizeof(cu_linear));
  ... // skipped 18 lines of code
  if(ret->count==0) return NULL;
```

If the sum of the two linear expressions passed as arguments becomes constant, the function returns `NULL` without freeing the memory allocated for the local variable `ret`. CUTE constructed this scenario automatically at the time of testing. Specifically, CUTE constructed the sequence of function calls `l1=cu_linear_create(0); l1=cu_linear_create(0); l1=cu_linear_negate(l1); l1=cu_linear_add(l1,l2,1);` that exposes the memory leak that valgrind detects.

4.2 SGLIB Library

We also applied CUTE to unit test *SGLIB* [25] version 1.0.1, a popular, open-source C library for generic data structures. The library has been extensively used to implement the commercial tool Xrefactory. SGLIB consists of a single C header file, `sglib.h`, with about 2000 lines of code consisting only of C macros. This file provides generic implementation of most common algorithms for arrays, lists, sorted lists, doubly linked lists, hash tables, and red-black trees. Using the SGLIB macros, a user can declare and define various operations on data structures of parametric types.

The library and its sample examples provide verifier functions (can be used as `repOk`) for each data structure except for hash tables. We used these verifier functions to test the library using the technique of `repOk` mentioned in Section 3.6. For hash tables, we invoked a sequence of its function. We used CUTE with bounded depth-first search strategy with bound 50. Figure 10 shows the results of our experiments.

We chose SGLIB as a case study primarily to measure the efficiency of CUTE. As SGLIB is widely used, we did not expect to find bugs. Much to our surprise, we found *two bugs* in SGLIB using CUTE.

The first bug is a segmentation fault that occurs in the doubly-linked-list library when a non-zero length list is concatenated with another zero-length list. CUTE discovered the bug in 140 iterations (about 1 seconds) in the bounded depth-first search mode. This bug is easy to fix by putting a check on the length of the second list in the concatenation function.

The second bug, which is a more serious one, was found by CUTE in the hash table library in 193 iterations (in 1 second). Specifically, CUTE constructed the following valid sequence of function calls which gets the library into *an infinite loop*:

```
typedef struct ilist { int i; struct ilist *next; } ilist;
ilist *htab[10];
main() {
  struct ilist *e,*e1,*e2,*m;
  sglib_hashed_ilist_init(htab);
  e=(ilist *)malloc(sizeof(ilist)); e->next = 0; e->i=0;
  sglib_hashed_ilist_add_if_not_member(htab,e,&m);
  sglib_hashed_ilist_add(htab,e);
  e2=(ilist *)malloc(sizeof(ilist)); e2->next = 0; e2->i=0;
  sglib_hashed_ilist_is_member(htab,e2); }
```

where `ilist` is a `struct` representing an element of the hash table. We reported these bugs to the SGLIB developers, who confirmed that these are indeed bugs.

Name	Run time in seconds	# of Iterations	# of Branches Explored	% Branch Coverage	# of Functions Tested	OPT 1 in %	OPT 2 & 3 in %	# of Bugs Found
Array Quick Sort	2	732	43	97.73	2	67.80	49.13	0
Array Heap Sort	4	1764	36	100.00	2	71.10	46.38	0
Linked List	2	570	100	96.15	12	86.93	88.09	0
Sorted List	2	1020	110	96.49	11	88.86	80.85	0
Doubly Linked List	3	1317	224	99.12	17	86.95	79.38	1
Hash Table	1	193	46	85.19	8	97.01	52.94	1
Red Black Tree	2629	1,000,000	242	71.18	17	89.65	64.93	0

Figure 10: Results for testing SGLIB 1.0.1 with bounded depth-first strategy with depth 50

Figure 10 shows the results for testing SGLIB 1.0.1 with the bounded depth-first strategy. For each data structure and array sorting algorithm that SGLIB implements, we tabulate the time that CUTE took to test the data structure, the number of runs that CUTE made, the number of branches it executed, branch coverage obtained, the number of functions executed, the benefit of optimizations, and the number of bugs found.

The branch coverage in most cases is less than 100%. After investigating the reason for this, we found that the code contains a number of assert statements that were never violated and a number of predicates that are redundant and can be removed from the conditionals.

The last two columns in Figure 10 show the benefit of the three optimizations from Section 3.5. The column OPT 1 gives the average percentage of executions in which the fast unsatisfiability check was successful. It is important to note that the saving in the number of satisfiability checks translates into an even higher relative saving in the satisfiability-checking time because `lp_solve` takes much more time (exponential in number of constraints) to determine that a set of constraints is unsatisfiable than to generate a solution when one exists. For example, for red-black trees and depth-first search, OPT 1 was successful in almost 90% of executions, which means that OPT 1 reduces the number of calls to `lp_solve` an order of magnitude. However, OPT 1 reduces the solving time of `lp_solve` more than two orders of magnitude in this case; in other words, it would be infeasible to run CUTE without OPT 1. The column OPT 2 & 3 gives the average percentage of constraints that CUTE eliminated in each execution due to common sub-expression elimination and incremental solving optimizations. Yet again, this reduction in the size of constraint set translates into a much higher relative reduction in the solving time.

5. RELATED WORK

Automating unit testing is an active area of research. In the last five years, over a dozen of techniques and tools have been proposed that automatically increase test coverage or generate test inputs.

The simplest, and yet often very effective, techniques use random generation of (concrete) test inputs [4, 8, 10, 20, 21]. Some recent tools use bounded-exhaustive concrete execution [5, 12, 29] that tries all values from user-provided domains. These tools can achieve high code coverage, especially for testing data structure implementation. However, they require the user to carefully choose the values in the domains to ensure high coverage.

Tools based on symbolic execution use a variety of approaches—including abstraction-based model checking [1,3], explicit-state model checking [27], symbolic-sequence exploration [22, 30], and static analysis [9]—to detect (potential) bugs or generate test inputs. These tools inherit the incompleteness of their underlying reasoning engines such as theorem provers and constraint solvers. For example, tools using precise symbolic execution [27, 30] cannot analyze any code that would build constraints out of pre-specified theories, e.g., any code with non-linear arithmetic or array indexing with non-constant expressions. As another example, tools based on predicate abstraction [1,3] do not handle code that depends on complex data structures. In these tools, the symbolic execution proceeds separately from the concrete execution (or constraint solving).

The closest work to ours is that of Godefroid et al.'s directed automated random testing (DART) [11]. DART consists of three parts: (1) directed generation of test inputs, (2) automated extraction of unit interfaces from source code, and (3) random generation of test inputs. CUTE does not provide automated extraction of interfaces but leaves it up to the user to specify which functions are related and what their preconditions are. Unlike DART that was applied to testing each function in isolation and without preconditions, CUTE targets related functions with preconditions such as data structure implementations. DART handles constraints only on integer types and cannot handle programs with pointers and data structures; in such situations, DART tool's testing reduces to simple and ineffective random testing. DART proposed a simple strategy to generate random memory graphs: each pointer is either `NULL` or points to a new memory cell whose nodes are recursively initialized. This strategy suffers from several deficiencies:

1. The random generation itself may not terminate [7].
2. The random generation produces only trees; there is no sharing and aliasing, so there are no DAGs or cycles.
3. The directed generation does not keep track of any constraints on pointers.
4. The directed generation never changes the underlying memory graph; it can only change the (primitive, integer) values in the nodes in the graph.

DART also does not consider any preconditions for the code under test. For example, in the oSIP case study [11], it is unclear whether some `NULL` values are actual bugs or false alarms due to violated preconditions. Moreover, CUTE implements a novel constraint solver that significantly speeds up the analysis.

Cadar and Engler proposed Execution Generated Testing (EGT) [6] that takes a similar approach to testing as CUTE: it explores different execution paths using a combined symbolic and concrete execution. However, EGT did not consider inputs that are memory graphs or code that has preconditions. Also, EGT and CUTE differ in how they approximate symbolic expressions with concrete values. EGT follows a more traditional approach to symbolic execution and proposes an interesting method that *lazily* solves the path constraints: EGT starts with only symbolic inputs and tries to execute the code fully symbolically, but if it cannot, EGT solves the current constraints to generate a (partial) concrete input with which the execution proceeds.

CUTE is also related to the prior work that uses backtracking to generate a test input that executes one given path (that may be known to contain a bug) [13,15]. In contrast, CUTE attempts to cover *all* feasible paths, in a style similar to systematic testing. Moreover, this initial work did not address inputs that are memory graphs. Visvanathan and Gupta [28] recently proposed a technique that generates memory graphs. They also use a specialized symbolic execution (not the exact execution with symbolic arrays) and develop a solver for their constraints. However, they consider one given path, do not consider unknown code segments (e.g., library functions), and do not use a combined concrete execution to generate new test inputs.

6. DISCUSSION

Our work shows that approximate symbolic execution for testing code with dynamic data structures is feasible and scalable. Moreover, we have shown how to efficiently generate dynamic data structures by incrementally adding and removing a node, or by aliasing two pointers. While we described an implementation for C, we have also developed an implementation for the sequential subset of Java. We are currently investigating how to test programs with concurrency using a similar method. We are also investigating the application of the technique to find algebraic security attacks in cryptographic protocols, and security breaches in unsafe languages.

Acknowledgements

We are indebtful to Patrice Godefroid and Nils Klarlund for their comments on a previous version of this paper and for suggestions on clarifying the relationship of the current work with DART. Moreover, the first author benefited greatly from interaction with them during a summer internship. We would like to thank Thomas Ball, Cristian Cadar, Sarfraz Khurshid, Alex Orso, Rupak Majumdar, Sameer Sundresh, and Tao Xie for providing valuable comments. This work is supported in part by the ONR Grant N00014-02-1-0715.

7. REFERENCES

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

[2] C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th International Conference on Computer Aided Verification*, pages 515–518, July 2004.

[3] 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.

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

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

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

[7] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proc. of 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

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

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

[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] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

[13] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proc. of the International Conference on Automated Software Engineering*, pages 219–227, 2000.

[14] 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.

[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] lp_solve. `http://groups.yahoo.com/group/lp_solve/`.

[18] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposia in Applied Mathematics.* AMS, 1967.

[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] Parasoft. Jtest manuals version 6.0. Online manual, February 2005. `http://www.parasoft.com/`.

[23] C. S. Pasareanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted java programs. In *Proc. of TACAS'01*, pages 284–298, 2001.

[24] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. Technical Report UIUCDCS-R-2005-2597, UIUC, 2005.

[25] SGLIB. `http://xref-tech.com/sglib/main.html`.

[26] Valgrind. `http://valgrind.org/`.

[27] 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.

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

[29] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[30] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems*, 2005.