Feedback-Directed Unit Test Generation for C/C++ using Concolic Execution

Pranav Garg*†, Franjo Ivančić*, Gogul Balakrishnan*, Naoto Maeda*‡, and Aarti Gupta*
*NEC Laboratories America, Princeton, NJ, USA
†University of Illinois at Urbana-Champaign, Urbana, IL, USA
‡NEC Corporation, Kanagawa, Japan

***Abstract*—In industry, software testing and coverage-based metrics are the predominant techniques to check correctness of software. This paper addresses automatic unit test generation for programs written in C/C++. The main idea is to improve the coverage obtained by feedback-directed random test generation methods, by utilizing concolic execution on the generated test drivers. Furthermore, for programs with numeric computations, we employ non-linear solvers in a lazy manner to generate new test inputs. These techniques significantly improve the coverage provided by a feedback-directed random unit testing framework, while retaining the benefits of full automation. We have implemented these techniques in a prototype platform, and describe promising experimental results on a number of C/C++ open source benchmarks.**

I. Introduction

Given the omnipresence of software in today's society, there is a great need to develop technologies that target effective verification technologies for software. In industry, software testing and coverage-based metrics are the predominant techniques to check correctness of software systems. This paper addresses automatic unit test generation for programs under analysis, in particular for programs written in C/C++.

In the past decade, there has been strong interest in developing practical techniques to generate tests that increase software coverage or discover faults that were previously hidden. These techniques often require the user to mark certain variables as test inputs, followed by an exploration of the related input space. Techniques such as random search or concolic execution [1], [2], [3] are often used to explore as many paths as possible based on these input variables.

On the other hand, there has also been progress on completely automated random test generation, e.g. the tool RANDOOP automatically generates test drivers for Java programs without user intervention [4]. Users of such tools include developers, quality assurance (QA) professionals, and acceptance testers of outsourced projects. An automated test generation approach has many benefits: it requires no input from the user, it does not require a deep understanding of the code, and it does not require manipulations of the code. RANDOOP is also well-suited to the needs of object-oriented programs.

RANDOOP relies on feedback-guided test drivers with randomly chosen values for test inputs. The most relevant feedback that RANDOOP considers is whether a test executes successfully. That is, RANDOOP classifies a randomly generated test driver by its execution behavior: If it executes without causing a visible failure, it is collected in a set that is used to generate other test drivers in the future. However, if it causes a failure (segmentation fault, assertion violation, thrown exception, ...), then we call such a test driver a *crash driver*. Note that crash drivers can indicate a real bug in the program, or they may violate some documented (or undocumented) object protocol thereby causing an expected program crash (not due to a bug in the program). These crash drivers are collected and presented to the user for further investigation.

Although directed-random test driver generation, such as implemented in RANDOOP, is generally very effective in the early stages of testing, it often reaches a coverage plateau [4]. It may not cover deep object interactions or deep branches using random methods and random test inputs. Another practical issue is that currently RANDOOP does not handle C/C++ programs. Due to the wide prevalence of C/C++ in industry, and the inherent semantic complexity of marrying powerful object-oriented abstraction mechanisms with potential low-level memory and performance optimizations, there is a stark need in industry for automated test generation for C/C++.

This paper follows the guiding principle of RANDOOP to generate test drivers without user intervention and without the user having to mark variables as test inputs. We propose a flexible testing framework that switches between directed-random search and concolic execution to automatically cover hard-to-reach portions of the program. We further improve concolic execution in this framework, by utilizing the unsatisfiable cores generated by SMT solvers to learn infeasible paths among related test cases. For programs with numeric computations, we selectively invoke non-linear constraint solvers when linearization of path queries contributes to infeasibility.

II. Overview of our Approach

We highlight the main components of our test generation framework in the algorithm shown in Figure 1. At the top level, we choose a test method by tracking the increase in coverage over time. We start with directed-random testing, à la RANDOOP, as described below. When it reaches a coverage plateau [4], we switch to concolic execution to cover new program regions. We use various parameters to control the effort spent, and can tune them to switch between the testing methods.

Directed-random test generation. We initiate our overall approach by creating unit tests in a manner similar to

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

Input: a set of C++ classes, associated contracts (if available), and a time-limit

Output: (T_N, T_X, T_R), where T_N, T_X and T_R are sets of test drivers

1: $T_N, T_X, T_R, C_S := \emptyset$ /* C_S: conflict sequences */
2: **while** time-limit not reached **do**
3: **if** `choose test method`=directed-random **then**
4: $m :=$ `choose target method`
5: $t :=$ extend-sequences (T_N, T_X, m)
6: **else** /* use concolic execution */
7: $b :=$ `choose target branch`
8: $\tau :=$`choose test driver`(T_N, T_X, b, C_S)
9: $\tau_S :=$ symbolize (τ)
10: $(\text{status}, s, c) :=$ concolic-execution (τ_s, b)
11: **if** status $=$ SAT **then** /* s contains solution */
12: $t :=$ instantiate τ_S with s
13: **else** /* UNSAT, c: unsatisfiable core */
14: **if** c is numerically symbolic **then**
15: $C_S := C_S \cup$ conflict-sequence(c)
16: **else** /* try ICP */
17: $\mathcal{I} :=$ concolic-icp-execution (τ_s, b) /* $\mathcal{I}$: set of solution boxes */
18: **if** $\mathcal{I} = \emptyset$ **then** /* ICP is UNSAT */
19: $C_S := C_S \cup$ executed path (τ_s, b)
20: **repeat** /* sample from solution boxes in $\mathcal{I}$ */
21: **if** need sampling in ICP solution boxes **then**
22: $i :=$ randomly sample from boxes in $\mathcal{I}$
23: $t :=$ instantiate τ_S with i
24: execute t and update sets T_N, T_X, T_R as needed
25: **until** sampling not needed **or** new test found **or** enough samples tried

Fig. 1. Algorithm for test generation framework

RANDOOP [4] (lines 4–5). A unit test typically consists of a sequence of method calls creating and mutating objects. These tests are built iteratively by choosing previously successfully executed tests, and combining them to form longer sequences targeting a particular method call [4]. The generated tests may also contain random values that are chosen for certain parameters of interest, such as parameters of type `int`. The generated test cases are then executed and automatically classified into one of the following three categories (line 24):

1) A set of test drivers T_R that cause a program runtime error, such as segmentation faults or assertion violations;
2) a set of test drivers T_X that end in uncaught exceptions at runtime; and
3) a set of test drivers T_N that exhibit normal runtime behavior, in that the driver does not match the description of the other categories.

The test drivers in the last category are regarded as normal because they have not exhibited any irregular behavior as yet. Implicit invariants of the program under analysis that may be violated but have not been explicitly checked (through embedded assertions, for example) are not tracked in the completely automated setup. The union of the sets T_R and T_X represents the crash drivers.

While deciding on the target method (line 4 of the algorithm shown in Figure 1), we bias our choice towards methods that have more unexplored branches. To do so, we maintain coverage metrics for each method. Once the target method is decided, its receiver object and its arguments are instantiated from previously executed unit tests. Method arguments of primitive types, like integers or floats, are instantiated with random values.

Concolic execution. Directed-random test generation often reaches a coverage plateau, which we wish to overcome utilizing *concolic execution*. Concolic execution improves upon the classical *symbolic execution* by partially substituting concrete execution values in cases where complex symbolic expressions arise or a symbolic expression becomes too coarse (such as calls to unknown functions). For any new input generated by concolic execution, we utilize the same classification of test drivers and add the so created tests to the set already found. After enough additional tests have been created, we return to the directed-random test generation method in the hope that it may extend the newly discovered tests even further. When initiating a concolic execution step, the main questions are:

- What is the coverage target?
- Which concolic execution should we consider?

To help make these selections, we maintain information on prior tests. We first select an uncovered branch as target (line 7). Next, with the help of prior runs, we choose a subset of current test drivers that are good candidates to reach the target branch (line 8). These test drivers contain certain parameters that are chosen randomly. We turn these randomly chosen values into symbolic variables (line 9), thereby yielding a concolic execution, which is then used to formulate an SMT problem to search for new input values. (This is similar to the *argument transformation* step in Evacon [5].)

More specifically, based on the concolic instrumentation, we create a path formula with a target branch check, for which the SMT solver may find a new input that can lead us to take that branch (line 10). If the SMT solver returns a value that satisfies the formula, we execute the newly generated test driver. Then, we add it to one of the three categories of test drivers, based on the execution outcome of the new test (line 24). However, if the SMT solver returns unsatisfiable, then there is no possible value for the symbolic variables that will let us reach the target branch, given the current instrumented concolic path.

Unsatisfiable core analysis in concolic execution. Note that the concolic instrumentation may restrict some program variables to be concrete. In particular, we are interested in concretization of non-linear arithmetic statements. In such cases, when the SMT solver returns an unsatisfiable core, we analyze the core to determine whether it contains only those parts of the execution trace that were numerically symbolic, i.e. without partial concretization (line 14). In such a case, we can use the unsatisfiable core in the future. In particular, it allows us to quickly decide which test driver cannot be extended to hit a target branch based on prior such unsatisfiable answers.

On the other hand, if the unsatisfiable core contains elements of partial concretizations, the target branch may be

reachable by using the same path but with a different input. The test drivers generated by RANDOOP contain many different concrete heap object "shape relationships". In this work, we use these concrete shape relationships but focus on partial concretizations due to non-linear computations. To decide whether it is infeasible to execute the target branch on the current path, we consult solvers that handle non-linear arithmetic. We use two different solving frameworks: CORAL [6] and interval constraint propagation (ICP). Note that CORAL has recently integrated ICP internally as well [7].

In Figure 1 we only show the use of ICP (line 17), since it requires some adaptation of the flow. Should ICP report that the path is unsatisfiable, we remember this information for future use. Often, however, ICP returns possible solutions for the input variables. While concolic execution queries resolved by SMT solvers (including CORAL) return one input vector, ICP returns sound candidate solution boxes. However, some solutions may not generate executions that reach the target branch. Hence, we randomly sample within the solution boxes to find inputs for new test runs (line 22). If an input results in an execution that reaches our target branch, we store the test driver in one of the three categories and continue (line 24).

This approach allows us to find tests for programs with complex non-linear arithmetic, which is usually not handled well by directed-random test driver generators such as RANDOOP. Note, that recently the *symbolic execution* engine of Java PathFinder (JPF) [8] was extended to handle complex, non-linear mathematical constraints using CORAL. As discussed above, we try to limit the calls to the non-linear solver to avoid high cost. We follow the *concolic execution* approach of favorizing linear path queries, and only call the non-linear solver if partial concretization was found using a light-weight unsatisfiable core analysis. Since JPF is generally applied to embedded software such as from the aerospace domain, it may be appropriate to always query more complex solvers. In our setting, where we generate tests for arbitrary domains, we show that a two-stage approach delaying calls to non-linear solvers has the potential to improve performance.

Handling of C/C++. The C/C++ programming language provides convenient abstraction and data encapsulation mechanisms for software developers. Such mechanisms include function and operator overloading, constructors and destructors, multiple class inheritance, dynamic virtual-function dispatch, templates, exceptions (where function specifications of thrown exceptions are discouraged), functors, and standard libraries such as STL and BOOST. However, these features also complicate analysis and testing of programs. Furthermore, due to the intrinsic complexity of mixing object-oriented programming on top of full-fledged C code, there is a need in industry to automate test generation for C/C++.

We have implemented the proposed test generation approach using CILpp, which is an in-house C/C++ infrastructure based on CiL [9]. CILpp models all aspects of C++ including templates, multiple inheritance [10], exceptions [11], C++ strings [12], etc. In contrast to other languages that RANDOOP has been applied to before, C/C++ also allows low-level memory constructs that do not follow a clean lifetime cycle such as construction, use/modification, and destruction. For example, it is common for large C/C++ projects to contain both complicated class hierarchies, as well as C structures/records called `structs`. For the initialization of such structures, we capture two common programming initialization idioms:

1) Record creation methods that allocate memory and initialize it appropriately, and pass the created memory region as the return pointer value back to the calling context. As an example, consider the file operation method `fopen`. This can be viewed as a constructor method for C records, and is thus easily adapted in the C/C++ framework.
2) Another common idiom is to have initialization methods that assume that memory allocation is handled separately. While this is possible in C++ class hierarchies, it is relatively uncommon. However, this is quite common in C programs, and we had to adapt RANDOOP to handle these cases. Figure 2 showcases a well-formed automatically generated object initializer for one of our evaluation benchmarks discussed in Section VII.

One additional complexity due to the low-level memory management capabilities of C is that we often need to account for correlations between method parameters that relate the length of a buffer to an associated pointer. A well known example is in the handling of C strings, for example for methods like `strncpy`.

Finally, since RANDOOP was initially developed for the *managed programming language* Java, we need to significantly adapt the test generation procedure for a language such as C/C++ that also allows low-level memory manipulations. For example, buffer overruns in Java are automatically caught and an appropriate exception is thrown. Similarly, the Java garbage collector removes the need to deallocate memory. However, we are also interested in capturing such bugs. Therefore, we use the dynamic checking tool *Valgrind* [13]. Note that the generated test drivers tend to be relatively small with negligible runtimes. We omit further details about the handling of C/C++ features for the sake of brevity.

```
board_t board;
memset(&board,0,
   sizeof(board_t));
board_start(&board);
```

Fig. 2. A generated initialization sequence for variables of type `board_t` in `gnuchess`

Contributions. The major contributions of this paper are:

- ❒ We propose a flexible testing framework that supports directed random unit test generation and concolic execution to target improved coverage. The concolic executor is applied on a test driver chosen from a pool of previously generated deemed "good" and extensible test drivers.
- ❒ We propose use of a light-weight unsatisfiable core analysis in case a path formula to a target branch is not satisfiable. This decides whether concretizations of arithmetic computations are relevant, or learns a conflict sequence for the path for future use. If the light-weight unsatisfiable core analysis finds that partial concretizations of the arithmetic computations contributed to the

```
class Base {
  Base();
  Base *foo(int i);
  ...
}
```

```
class Derived :
  public Base {
    Derived(int i);
    void bar(Base &b);
    ...
}
```

Fig. 3. Small source code snippet representing classes under test

```
Base b0();
```

```
Derived d0(1);
```

```
Derived d0(1);
Base *b1=d0.foo(0);
```

Fig. 4. Sequences worth extending generated by RANDOOP

unsatisfiability, we propose to use non-linear solvers lazily to find solutions.

❒ Finally, we have implemented the techniques in a fully automated unit test generation tool, that handles programs written in C/C++. The tool handles all complex semantical features of C/C++, including exceptions and multiple inheritance, using our in-house `CILpp` framework [10]. We present experimental evidence and an evaluation of our approach showing promising results.

III. BACKGROUND

A. Feedback-Directed Random Unit Test Generation

We initiate our overall test generation method by creating tests in a manner similar to RANDOOP [4]. An object-oriented test typically consists of a sequence of methods calls creating and mutating objects, followed by some kind of consistency check. These tests are built iteratively by choosing previously successfully executed tests, and combining them to form longer sequences targeting a particular method call [4]. We use the code snippet in Figure 3 to illustrate this process.

RANDOOP randomly chooses methods to target, and supplies formal arguments for parameters based on previously created sequences, that are deemed to be "worth extending". A sequence is deemed worth extending, if it has not yet produced a runtime violation. Traces that produce runtime violations are collected separately as evidence of potential bugs.

To create an executable test driver from a sequence, it is placed into a `main` function. Test drivers may be terminated using sanity checks. The sequences produced by RANDOOP generate formal arguments to match the corresponding API chosen as a target. When parameters of primitive type are needed, it generates random inputs of the correct type, as shown in Figure 4 for `int` arguments.

In order to extend a sequence, RANDOOP chooses a new target function to call. Suppose that we are interested in generating a call to `Derived::bar`. Four possible extensions from the set shown in Figure 4 are shown in Figure 5. Note that the third generated sequence may fail in the test driver, due to variable `b1` potentially being `NULL`. The most useful feedback that RANDOOP utilizes is the distinction whether a test case is deemed worth extending.

B. Concolic Execution

Let P be a program over variables $X = \{x_1, \cdots, x_m\}$ and let its CFG be represented as $\Pi = (X, N, E, n_o, n_e)$ where

```
Base b0() ;
Derived d0(1);
d0.bar(b0);
```

```
Derived d0(1);
d0.bar(d0);
```

```
Derived d0(1);
Base *b1 =
  d0.foo(0);
d0.bar(*b1);
```

```
Derived d0(1);
Base *b1 =
  d0.foo(0);
if (b1)
  d0.bar(*b1);
```

Fig. 5. Four extended sequences generated by RANDOOP

N is the set of nodes, $E \subseteq N \times N$ is the set of edges and $n_o, n_e \in N$ are unique entry and exit nodes. The set E is a disjoint union of edges E_a and E_c labeled with assignment and conditional statements. Let $Exp : Exp\ op\ Exp \mid X$, $op = \{+, -, \times\}$ represent the grammar of expressions over X. We assume assignments are of the form $s_a : x_i \leftarrow e,\ e \in Exp$ and conditional statements have guards $s_c : e \sim 0$, for $\sim = \{<, \leq, >, \geq, =, \neq\}$. For simplicity, we assume that the program does not have pointers, arrays or heap manipulation. Note, however, that our implementation handles these features.

Let $\pi : n_o, n_1, \cdots, n_k, n_e$ be a path of the CFG. Concolic execution is used to obtain a logical constraint φ on initial values of variables X such that the program execution is forced along the desired path π. To accomplish this, the program is instrumented to track the constraints as it executes. Specifically, for a path prefix $\pi_i : n_o, n_1, \cdots, n_i$ of π, the instrumentation code tracks (φ_i, F_i) where φ_i represents the path constraint computed for π_i and $F_i : X \rightarrow Exp$ represents a map from variables x_i to their current values expressed as a symbolic expression over the initial values of X. The map F_i is extended to expressions over X in a natural way. To begin with $\varphi_0 = \mathit{true}$ and $F_0(x_i) = x_i$ for all $x_i \in X$. For each statement edge $n_{i-1} \rightarrow n_i$, the state (φ_{i-1}, F_{i-1}) is updated to (φ_i, F_i) as follows:

- **Assignment:** Consider the assignment statement $s_a : x_i \leftarrow e$. The instrumentation updates the symbolic information $(\varphi, F) \xrightarrow{s_a} (\varphi, F')$ where F' is:

$$F'(x_j) = \begin{cases} e[F(x_1)/x_1, \cdots, F(x_m)/x_m] & j = i \\ F(x_j) & j \neq i. \end{cases}$$

- **Condition:** For a condition $s_c : e \sim 0$, we update the symbolic information $(\varphi, F) \xrightarrow{s_c} (\varphi', F)$ where

$$\varphi' = \varphi \wedge \varphi^i, \varphi^i = (F(e) \sim 0).$$

The generated constraint φ_i typically contains only linear arithmetic by concretizing non-linear expressions; i.e., if there is any non-linear expression, symbolic expressions $F(x_i)$ corresponding to the relevant variables are substituted by their concrete values observed during the actual concrete execution. Concretization is also used to approximate other program features, such as return values of external function calls. The constraint φ_i thus under-approximates the input values for variables X that force execution along path prefix π_i.

IV. INTEGRATION OF TEST GENERATION METHODS

Directed random test generators allow automatic generation of tests for object-oriented programs. However, once this approach saturates, we use concolic techniques for increasing

the structural coverage of the program under test. In this phase, from the existing pool of test sequences, concolic execution is used to explore uncovered program paths. After the concolic phase has run for some time, the new test sequences thus generated are used to seed further directed random test generation. Hence, we shift back and forth between random test generation and concolic execution in a symbiotic manner.

For each conditional statement, we pair the *then*- and *else*-branches of the *if*-block. For concolic execution we pick a branch that has not been covered before, but whose other branch in the pair has been covered by some test case. We also restrict our search to branches whose guard on concolic execution is symbolic. We use a simple dataflow analysis to obtain an over-approximation of such *symbolic branches*.

Given a target branch and a test sequence, we use concolic execution to explore the target. To do so, we inspect all method invocations in the test and automatically introduce symbolic variables in place of method arguments with primitive types. Thus, we obtain a test sequence which is symbolic in these inputs. As an example consider the symbolic test sequence in Figure 6 obtained from the concrete test of Figure 4. Test cases obtained by concolic execution are added to the pool. After running concolic execution for a set of targets, we switch back to randomized test generation to iteratively generate longer test sequences, thereby providing an amplification in coverage.

```
int symbolic1 = 1 ;
havoc(&symbolic1) ;
Derived d0(symbolic1);
int symbolic2 = 0 ;
havoc(&symbolic2) ;
Base *b1 =
  d0.foo(symbolic2) ;
```

Fig. 6. A symbolic test sequence

Example 4.1 (Coverage improvement): We observed a marked improvement by combining concolic execution with directed random test generation for certain benchmarks. A snippet of a method `Triangle::intersect` is shown in Figure 7. In the randomly generated test cases, the concrete values for `du0du1` and `du0du2`, computed at the lines marked `mult-1` and `mult-2`, were positive. Hence, the test cases returned at the statement marked `early-return`. Using concolic execution, inputs for a new test case were generated such that `du0du1` becomes negative, thus avoiding the `early-return` statement and exploring the following portion of the method. This single generated test extended the branch coverage by 6.1% due to the sequence of branches in the remainder of the execution. The generated test case was then further extended by random test generation in a follow-up iteration of our approach. This resulted in additional 7.5% branch coverage. Thus, by leveraging concolic execution within the random test generation framework, these tests were able to improve branch coverage by 13.6%. □

V. Unsatisfiable Cores in Concolic Executions

Tests generated using concolic execution have the same code structure as their parent tests, but only differ in their inputs. Any two tests generated from the same parent often share large portions of their execution trace. When a path in a particular test driver to a target branch is infeasible, we use unsatisfiable

```
bool Triangle::intersect(const Triangle &t)
{
  ...
  du0=(N1[0]*U0[0]+N1[1]*U0[1]+N1[2]*U0[2])+d1;
  du1=(N1[0]*U1[0]+N1[1]*U1[1]+N1[2]*U1[2])+d1;
  du2=(N1[0]*U2[0]+N1[1]*U2[1]+N1[2]*U2[2])+d1;
  if(fabs(du0)<0.000001) du0=0.0;
  if(fabs(du1)<0.000001) du1=0.0;
  if(fabs(du2)<0.000001) du2=0.0;
  du0du1=du0*du1;            /* mult-1 */
  du0du2=du0*du2;            /* mult-2 */
  if(du0du1>0.0f && du0du2>0.0f)
    return false;            /* early-return */
  ...
```

Fig. 7. Code snippet from `coldet` showing use of concolic execution

cores to generalize the reason of infeasibility of the path. These unsatisfiable cores can be used in the future to rule out infeasible paths during concolic execution of related test cases. As noted in [14], symbolic execution is many times slower than executing a program. The use of unsatisfiable cores helps by preventing redundant SMT queries.

A. Using Unsatisfiable Cores for Finding Conflict Sequences

We use the core of an unsatisfiable formula for extracting structural reasons for the infeasibility of the corresponding path. This is achieved by semantically annotating constraints obtained in the concolic execution with a set of responsible control flow edges. Let π be a path in the CFG $\Pi = (X, N, E, n_o, n_e)$ chosen for concolic execution. We modify the concolic execution described earlier to also track the CFG edges in path π that contribute to each conjunct φ_i in the constraint φ. Formally, apart from $(\varphi : \bigwedge_{i\in I} \varphi_i,\ F : X \to Exp)$, the result of concolic execution is augmented with maps $c : I \to 2^E$ and $f : X \to 2^{E_a}$. Here, c maps each conjunct φ_i in φ to a set of *contributing* edges in π, and similarly f maps each variable $x_i \in X$ to a set of assignment edges in π which lead to its symbolic expression $F(x_i)$. The maps c and f are updated as follows:

- **Assignment:** Consider the assignment edge e_a labeled with statement $s_a : x_i \leftarrow e$. The instrumentation updates the symbolic information $(\varphi, F, c, f) \xrightarrow{s_a} (\varphi, F', c, f')$ where F' is the same as described earlier and
$$f'(x_j) = \begin{cases} \{e_a\} \cup \bigcup_{x_k \in e} f(x_k) & j = i; \\ f(x_j) & j \neq i. \end{cases}$$
- **Condition:** Consider the edge e_c with the guard $s_c : e \sim 0$. The instrumentation code updates the symbolic information $(\varphi, F, c, f) \xrightarrow{s_c} (\varphi', F, c', f)$ where $\varphi' = \varphi \wedge \varphi_i$, $\varphi_i = (F(e) \sim 0)$ and
$$c'(j) = \begin{cases} \{e_c\} \cup \bigcup_{x_k \in e_c} f(x_k) & j = i; \\ c(j) & j \neq i. \end{cases}$$

The augmented information (c, f) allows us to compute a conflict sequence of edges by tracking the set of edges that contribute to each conjunct in the path constraint φ.

Definition 5.1 (Conflict Sequence): Let (φ, F, c, f) be the result of concolic execution of π and let φ be unsatisfiable

```
float x=ext1();//assume x=1
float y=ext2();//assume y=1
float z = 0.0 ;
if ( y > 0.0 )
  z=x*y; //concolic: z=1*y
assert(z>=0.0);//target
```

```
real x = *; //p1
real y = *; //p2
real z = 0; //p3
assume(y>0);//p4
z = y ;     //p5
assert(z<0);//p6
```

Fig. 8. Light-weight infeasible core analysis: The infeasible core contains the conjuncts `p4`, `p5` and `p6`. The Boolean marking function ρ evaluates to true for `p5`. Thus, we decide to invoke a non-linear solver.

with an unsatisfiable core given by index set I. The corresponding conflict sequence C is the set of edges $\bigcup_{i \in I} c(\varphi_i)$, contributing to the conjuncts in the unsatisfiable core, in the order of appearance in π.

We use conflict sequences corresponding to an infeasible path to avoid potentially expensive checks for other paths. A conflict sequence C deduces the infeasibility of a program path π, if C is a subsequence of the edges occurring in π. Additionally, any variable that is read by an edge $e \in C$, must not be defined by the edges in π not belonging to C; i.e. the values flowing into the conflict edges C should be defined by other edges in C or should be the initial values. This latter check ensures that the edges in π which are not in C do not interfere with the conflict sequence. We conclude the infeasibility of π if there exists some compatible match with a conflict sequence. Note that this way of avoiding SMT queries is similar to the learning from proofs we used for static analysis [15], but the specific analysis to derive the conflict sequence is different here.

B. Light-Weight Infeasible Core Analysis

The process of matching paths with conflict sequences is presented for a language without pointers, calls to external functions, or non-linear computations. The concolic execution of a statement outside this scope substitutes (a part of) the symbolic expressions involved with concrete values. Due to these concretizations, it is possible that a path π compatible with a conflict sequence is actually feasible. So, we modify our approach slightly: Given a test driver and an instantiation of its symbolic inputs, if the program path π is infeasible, we obtain the conflict sequence C corresponding to the unsatisfiable core of the path constraint. If C does not contain any edge with concretization, we store C. If, however, C contains an edge with concretization due to linearization, the infeasibility of π might be due to the particular concrete values used. In such a case, we re-analyze the path using a non-linear solver.

In addition to the augmented information (c, f) discussed before, we also add a Boolean marking $\rho : I \to \{0, 1\}$, which tracks whether a conjunct φ_i is partially concretized in its arithmetic computations. This is used to quickly decide whether to try the non-linear solver for a less concretized path formula. An example of this analysis is shown in Figure 8.

VI. ICP and Symbolic Arithmetic Execution

ICP algorithms have been applied to various non-linear computing problems involving thousands of variables and constraints [16]. ICP differs from other numerical solution-finding algorithms in the following reliability guarantees:

- ICP always terminates, returning either "unsatisfiable", or "satisfiable" with an interval over-approximation of a solution (or the solution set).
- When ICP returns "unsatisfiable", it is always correct.
- When ICP returns "satisfiable", the solution may be spurious; but its error is within a user-specified bound.

Given a set of real constraints and interval bounds on their variables, ICP [17] successively refines an interval over-approximation of its solution set by narrowing down the possible value ranges for each variable. ICP either detects the unsatisfiability of a constraint set when the interval assignment on some variable is narrowed to the empty set, or returns interval assignments for the variables that tightly overapproximate the solution set, satisfying some preset precision requirement.

If the light-weight infeasible core analysis determines that partial concretizations due to non-linear computations contributed to the infeasibility, the target branch may be reachable using the same path. In this situation, we extend the program instrumentation to generate a concolic execution that concretizes heap objects but treats arithmetic computations as fully symbolic, even if computations are non-linear.

On the non-linear path formula, ICP may still return the answer *unsatisfiable*. Thus, lifting the partial concolic concretization performed due to non-linear numerical computations is not enough to generate test inputs that will reach the target branch.[1] Alternatively, ICP may return potential solution boxes to the constraints at hand that are not feasible. By increasing the precision requirement, that is, by allowing smaller maximal errors, we can reduce the number of such cases.

When ICP returns candidate solutions, we sample these to find test inputs for new tests. Sampling of interval solution boxes is cheap, since we can sample each input variable independently of other inputs. There are many known sampling strategies with good coverage properties. We perform a fixed-length sampling of the candidate solutions returned by ICP. Should a test input result in a program execution that reaches the target branch, we store the generated test driver.

```
real x=*,y=*,z=0;
assume(y>0) ;
z = x*y ;
assert(z<0) ;
```

Fig. 9. Non-linear path

Example 6.1: Figure 9 shows a concolic trace containing non-linear arithmetic based on Figure 8. Recall that standard concolic execution failed to find a test input in this case. ICP however finds an input where `x` is negative and `y` positive. □

Example 6.2 (Improved coverage due to ICP): In our experiments, we observed that numerically symbolic execution using ICP was successful in generating tests where directed random test generation and standard concolic execution failed. A code snippet of a method `Box::intersect` is shown in Figure 10. The targeted condition is `abs_cross[0] > f` is false, and thus control reaches the `return false;`

[1]Note that we currently use the ICP solver in an idealized arithmetic setting, performing computations in the domain of reals. Prior work has shown how to extend concolic execution to model floating-point computations [18], [19].

```
bool Box::intersect(const Vector3D& O,
                    const Vector3D& D) {
  ...
  Vector3D cross = CrossProduct(D,diff);
  abs_cross[0]=flabs(cross[0]); /*abs. value*/
  f = m_Size[1]*abs_segdir[2] +
      m_Size[2]*abs_segdir[1];
  if ( abs_cross[0] > f )   /* target-branch */
    return false;           /* target-block  */
```

Fig. 10. Snippet from `coldet` showing use of ICP-enabled concolic execution

statement. The numeric computation involves the comparison of an absolute value of a cross product with the sum of two multiplications. The concolic execution finds that the original concolic trace is unsatisfiable. However, after relaxing the constraint to keep the numerical computations symbolic, ICP finds a solution box. Sampling the solution box finds a new test input that allowed us to reach the `target-block`. □

VII. Experiments and Evaluation

We implemented the proposed approach in our in-house C/C++ infrastructure called `CILpp` [10]. We adapted RANDOOP [4] with respect to C++ semantics. For example, certain Java requirements are not present in C++, e.g. `equals` methods should be symmetric. C++ also has complexities due to memory-safety issues, object lifetimes, and others – these are handled by `CILpp`. We also lifted the concolic executor CREST [20] to `CILpp` and allowed tracing of floating-point operations. Finally, for non-linear queries, we integrated the ICP solver `RealPaver` [21] and the meta-heuristic constraint solver `CORAL` [6] into our framework. Experiments are performed with a fixed time bound for each investigation strategy.

A. Experimental Setup

We performed an evaluation using various open-source C/C++ projects. We pre-processed the source code and simplified the control structure so that branch conditions are atomic. Hence, the branch coverage that we present is akin to *condition/decision coverage* on the original program. We report results for the following projects: a red-black tree implementation, a linked list implementation, a priority queue implementation, `coldet`, a collision detection library used in gaming, `gnuchess`, a computer chess program, `ddrescue`, a data recovery tool, and `tinyXML`, a widely used XML parser. We report the effective LOC (which includes *relevant* header file instantiations) for the modules under test and the number of functions, statements and branches for each project in Table I. Note that `gnuchess` is divided into multiple modules – thus, we split the benchmarks into two parts.

In the following, we investigate the following *research questions* (RQ) in more detail:

1) Does concolic execution help our test generation framework improve upon pure directed random testing?
2) Are non-linear solving techniques such as interval constraint propagation useful to increase coverage when applied to feedback-directed generated test drivers?
3) Is our combined framework able to discover previously unknown bugs in C/C++ benchmarks?
4) Can our framework reveal new types of bugs previously not handled by RANDOOP, such as memory leaks?

TABLE I
STATISTICS ABOUT BENCHMARKS USED IN OUR STUDY

Benchmark	Lang.	ELOC	# func	# stmts	# branches
`tree`	C	1.6K	18	137	48
`list`	C++	1.5K	10	109	40
`queue`	C++	2.9k	18	296	102
`coldet`	C++	10.7K	118	2987	834
`gnuchess1`	C	29.3K	316	7291	2195
`gnuchess2`	C	40.2K	272	10022	3026
`ddrescue`	C++	19K	151	3549	1336
`tinyxml`	C++	13.4K	284	3950	1215

TABLE II
BRANCH COVERAGES FOR DIRECTED RANDOM TESTING (DR), CONCOLIC EXECUTION (CE), AND THE COMBINED TEST FRAMEWORK (CTF)

Bench-mark	Branch coverage			CTF improvement over	
	DR %	CE %	CTF %	DR	CE
`tree`	92	90	92	0%	2%
`list`	43	43	43	0%	0%
`queue`	65	67	69	6%	3%
`coldet`	41	28	75	83%	170%
`gnuchess1`	28	25	31	9%	23%
`gnuchess2`	14	11	30	110%	166%
`ddrescue`	21	22	37	76%	70%
`tinyxml`	43	43	44	1%	2%

B. Improving Coverage over Directed Random Testing

First, we investigate RQ1. The experimental results are summarized in Table II. The function level coverage is generally the same between directed random and our approach since we adopt the sequence extension mechanism from RANDOOP. As previously observed, random testing generally performs well for container classes [22]. However, for most other benchmarks our approach generally dominates pure directed random test generation and concolic execution in terms of branch coverage. Note that the branch coverages on some benchmarks seem low – this is largely due to the fact that we did not provide mock environments for system calls, that we did not provide local files that were required by some methods, and that many of these benchmarks follow a defensive programming style. For example, in `gnuchess` parameters are checked for validity at every function entry - even though the same checks have been performed previously on the same path. Thus, many condition branches cannot be reached (which we measure, as noted above).

We highlight our experimental results for one of the larger benchmarks (`coldet`) in Figure 11. We present the coverage results in terms of percentage of the complete code under analysis. As can be seen, with the help of directed random testing, we are able to quickly cover over 85% of all functions. The test sequence generator only generates calls to public methods, i.e. private methods need to be reached indirectly.

Figure 11 highlights the coverage improvement obtained by our test generation, in particular with respect to branch coverage. While pure directed random test generation reaches

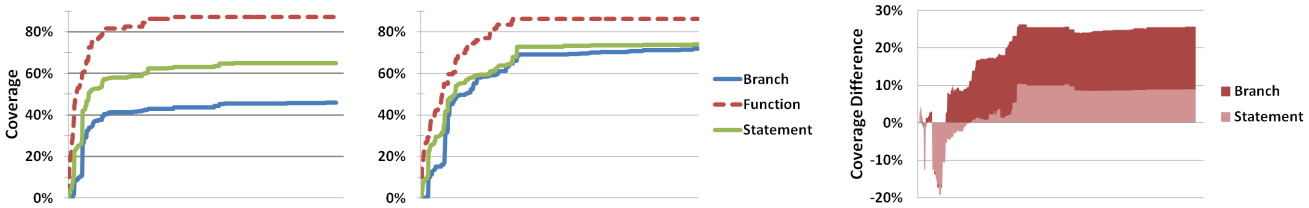

Fig. 11. Experimental results for our test generation framework for `coldet`: On the left, we show results of our RANDOOP implementation for branch, function and statement coverage over time using dark blue, dashed red, and light green lines, respectively. The middle shows the corresponding coverage over time for our combined framework. The graph on the right shows the difference between our combined framework and pure directed-random test generation for branch and statement coverage using dark red and light pink, respectively.

TABLE III
EXPERIMENTS REGARDING NON-LINEAR SOLVER HEURISTICS

Benchmark	LIA	CORAL	LIA+ICP	LIA+CORAL
coldet	27%	20%	58%	75%
gnuchess1	24%	29%	25%	31%

a coverage plateau at about 45% of branches, our approach generates tests that are able to cover over 70% of branches. This clearly highlights the benefit of adding concolic execution. Our approach also covers around 9% more statements than the tests generated using directed random testing only. As can be seen from the branch coverage over time, there are various points during test generation where intermediate plateaus are reached. However, since we switch to concolic execution early, these initial plateaus are quickly overcome and subsequent spikes are noticeable. It is also interesting that the tests may cover less branches and statements than pure directed random based tests in the beginning. This is due to the fact that we initiate concolic execution within some function before other functions have been exercised (see the dashed lines in Figure 11). The difference of called functions, however, converges to zero quickly, where deeper intra-method exploration using concolic execution shows its effectiveness.

Answer: The combination of concolic execution with random directed testing successfully alleviates some of the drawbacks of random directed testing alone.

C. Test Generation and Non-linear Path Queries

In this section we address RQ2, namely whether solving non-linear path queries helps to increase test coverage of general purpose programs. Table III highlights the coverage of generated tests using four different strategies: LIA uses a state-of-the-art linear integer arithmetic solver [23] only (without trying to find solutions to non-linear queries), CORAL always tries to answer queries using CORAL [6] without trying to first perform a linearization of the path query, LIA+ICP, where we use ICP to solve non-linear path queries after the initial linearized path query was found to be unsatisfiable due to partial concretization, and LIA+CORAL where we use CORAL instead of ICP when trying to solve the non-linear path query.

Table III highlights a number of things: First, (linear) concolic execution misses many reachable branches when compared to the other strategies. Secondly, using full arithmetic symbolic execution via CORAL tends to be expensive thus not reaching many branches in coldet. As can be seen especially in the column LIA+CORAL, performing a concolic analysis first, and lazily using more expensive queries has clear benefits. In our experiments, LIA+CORAL also clearly outperformed LIA+ICP. This is likely due to the fact that we did not optimize the error precision parameter of ICP, using a relatively large maximal error. This leads to fast query results, but with large solution boxes and we limited the sampling of these to too few samples.

TABLE IV
FAILURE INVESTIGATION FOR gnuchess

Reason	Unique	Total
Likely bugs	9	37
Implicit protocol	2	20
Unexpected parameters	16	16
Total	**27**	**73**

Answer: Selective use of non-linear solvers to resolve complex path queries can substantially increase code coverage of feedback-directed generated test drivers.

D. Analysis of Test Drivers Leading to Failures

In this section we investigate RQ3, namely whether our generated test drivers uncover serious bugs in C/C++ benchmarks such as segmentation faults, for example. We report our findings for some of the larger benchmarks.

gnuchess: Table IV shows the investigation result of crash drivers generated for the gnuchess project. We investigated 73 generated test drivers leading to crashes, of which 37 were judged to be due to bugs in gnuchess that could be resolved using 9 unique fixes. These bugs are mostly potential NULL pointer accesses, where calling contexts do not check for NULL pointer return values. The analysis also revealed a potential infinite loop, as well as a buffer underflow. In further 20 cases, we concluded that the test driver is likely violating some implicitly assumed event ordering (called *implicit protocol* in the table). Since the function call depth between the function called by the test driver and the offending program statement is quite large (about five in the two uniquely identified cases), we believe that the code should be improved. Finally, we found 16 test drivers that violated some assumed precondition on parameters of a called method. For example, in 13 cases, a NULL pointer was passed to functions, which did not expect such an argument. In another case, a constant string was passed to a function that tried to modify the string - violating its expectation as well. Furthermore, in two cases the test driver did not provide well-correlated parameters to a called function. In one such case, a format string was passed that included %s placeholders, but no additional string

```
Vector3D Normalized () const {
  return (1.0f/Magnitude())*(*this);
}
Plane(const Vector3D& a, const Vector3D& b,
      const Vector3D& c) {
  normal=CrossProduct(b-a,c-a).Normalized();
```

Fig. 12. Code fragment of `coldet`

argument was given to match this format string. We believe that for these 16 test drivers no source code fixes are needed.

`coldet`: We investigated a limited number of the generated crash drivers for the `coldet` benchmark. We found that two default constructors in `coldet` (for class `Triangle` and record `Matrix3D`) do not initialize member fields of the so created objects, thus causing several issues in many test drivers. These default constructors should either be declared `private` instead of the current designation as `public`, or should initialize member fields. We also found an instance of undefined behavior in `coldet` caused by a bad cast from a `NaN` (*not-a-number* floating-point value) to an integral type. The `NaN` value was traced back to a well-defined floating-point division-by-zero in the code fragment shown in Figure 12. The test driver generated input vectors such that normalization is performed on a 3-dimensional zero-vector, thus generating a 3-dimensional vector where each direction stored as a floating-point becomes a `NaN` (note that `Magnitude()` on a zero-vector returns zero). A variety of fixes can be designed – one option would be to abort the `Vector3D::Normalized()` call with an exception, when performed on a zero-vector.

`tinyXML`: We also analyzed a subset of the generated crash drivers in `tinyXML`. We found a variety of bugs, including a potential NULL pointer dereference in the `TiXmlAttribute::Next` and `TiXmlComment::Parse` methods, a double-free (through two calls to the destructor) of an object, if it is allocated on the stack, a stack overflow issue in the destructor of `TiXmlNode`, and multiple infinite loops.

Answer: The analysis of the failure inducing test drivers for a variety of benchmarks shows that our approach successfully generates interesting bug-revealing tests.

E. Finding Memory Leaks using Generated Test Drivers

In this section we investigate RQ4. We automatically investigated nearly 600 generated test drivers that were deemed as good test drivers worth extending for possible memory leaks in `coldet`. In order to report memory leaks in the benchmark only and not in the test driver, we need to de-allocate any memory that was created in the test driver using the various allocation methods such as `new`, `new[]`, `malloc`, etc. Thus, we alter the generated test drivers by adding the appropriate de-allocation calls to the end of the test driver in reverse order. Then, we investigated the test drivers using *Valgrind* [13] and inspected the reported leaks. We discovered two related memory leaks in the `BoxTreeInnerNode::createSons` method, which overwrites two allocated member fields without freeing the associated memory first. Note that the tool LEAKPOINT [24] could be used to find the location of the bug automatically.

We also analyzed the test drivers for some other benchmarks. However, the test drivers did not reveal memory leaks in `gnuchess`, for example. The `gnuchess` program assumes an a priori allocated chess board which is allocated and de-allocated after its use in the test driver. No internal allocations are made and thus no memory leak is possible there. Test drivers for some other benchmarks did not reveal additional memory leaks, either. We believe that the generated tests may be missing some memory leaks, although we are not certain of that fact. To increase the chance of finding additional memory leaks we may need to add heuristics that target paths with missing de-allocations or with overwriting of previously allocated memory. Forcing repetitive calls to methods that include allocations may also help, for example.

Answer: Our current test generation framework helps find a limited number of memory leaks. However, some memory leaks have possibly not been revealed. Future research potential exists in discovering appropriate search heuristics.

F. Threats to Validity

Threats to external validity. We evaluated our approach for automated test generation of C/C++ programs for a variety of benchmarks. However, we cannot guarantee that the set of benchmark is representative of all domains. To mitigate this limitation, we strived to include some that heavily rely on arithmetic computations and others that do not.

Threats to internal validity. One internal validity threat is the correctness of our implementation to generate test drivers for C/C++. We rely on the correctness of our `CILpp` framework for C/C++, which has been thoroughly tested and is in production use for program analysis within NEC in a tool called VARVEL [25]. We have also inspected the newly developed code for correctness. Finally, we have tested the implementation using *Valgrind* by comparing generated test drivers with and without `CILpp`, and our code coverage and symbolic execution instrumentation (simply relying on standard g++ compilers) to eliminate bugs in the tool-chain.

Threats to construct validity. We used coverage metrics on the program based on the executed test drivers to evaluate the effectiveness of our approach. To mitigate the issue with such threats, we used standard coverage metrics used in related studies. Furthermore, we relied on the well-studied CREST [20] tool to generate most of our coverage metrics and believe that the reported numbers are accurate and adequate.

VIII. Related Work

RANDOOP automatically generates unit tests for Java programs without user intervention [4]. In [26], static analysis has been used to improve upon it. RANDOOP has recently also been extended to test multithreaded code [27]. Concolic execution for software testing was first proposed in [1], [2] and subsequently used in many testing tools [3], [14], [28]. Recent tools, such as KLEE [3], can generate tests fully automatically, in limited circumstances, by allowing fuzzing over the input arguments to the `main` function. More recently [29] provides

an approach, alternate to concolic execution, for mixing symbolic execution with concrete-symbolic solving.

For industrial-strength object-oriented programs though, scalability is still a challenge in applying symbolic execution. Apart from the early depth first search strategy in concolic execution [1], other path exploration strategies have been proposed, such as hybrid concolic search [30], random paths search [20], generational search [14] and pruning the search space using state comparison [31]. However, symbolic execution is many times slower than executing the program. Thus, many approaches analyze test cases for all possible combinations of method sequences up to a bounded length only. One way of mitigating the symbolic execution cost is to distribute the test generation queries [32], [33], [34].

Recently, a number of approaches have been proposed that combine concolic execution with other testing strategies. One such method is hybrid concolic search [30] which combines pure random test input generation with concolic execution for user-marked input variables. More closely related to our work is Evacon [5]. Evacon combines evolutionary testing with concolic execution to achieve higher coverage of class unit tests. In Evacon, generated tests are also automatically symbolized, and presented to the concolic executor. However, as the experiments in [5] suggested, the evolutionary testing approach is expensive when compared to RANDOOP. RANDOOP was basically able to cover as many branches as Evacon, except in one degenerate case. We believe that the advantage of RANDOOP is that it easily scales to much larger units of testing, as shown in [4]. Thus, we believe that adding concolic execution to a RANDOOP-style test generation is more favorable when compared to evolutionary methods. Finally, note that Evacon [5] did not utilize the unsatisfiable core for non-linear solvers to find relevant test inputs, when the standard concolic executor fails.

IX. CONCLUSIONS

This paper presented a fully automatic approach for generating tests for object-oriented programs written in C/C++, using a feedback-directed random testing with concolic execution to overcome coverage plateaus. We also presented how an unsatisfiable core analysis can be used to guide the automatic test generation framework forward, and used non-linear solvers lazily to find inputs on complex path queries. Finally, an implementation of the approach has been presented, with encouraging experimental results.

REFERENCES

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*. ACM, 2005, pp. 213–223.

[2] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/SIGSOFT FSE*. ACM, 2005, pp. 263–272.

[3] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX Association, 2008.

[4] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*. IEEE Computer Society, 2007.

[5] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *ASE*, 2008.

[6] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "CORAL: Solving complex constraints for Symbolic PathFinder," in *NASA Formal Methods*, ser. LNCS, vol. 6617. Springer, 2011, pp. 359–374.

[7] M. Borges, M. d'Amorim, S. Anand, D. H. Bushnell, and C. S. Păsăreanu, "Symbolic execution with interval solving and meta-heuristic search," in *ICST*. IEEE, 2012, pp. 111–120.

[8] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *ISSTA*. ACM, 2004, pp. 97–107.

[9] 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*, 2002.

[10] J. Yang, G. Balakrishnan, N. Maeda, F. Ivančić, N. Sinha, A. Gupta, S. Sankaranarayanan, and N. Sharma, "Object model construction for multiple inheritance in C++ and its applications to program analysis," in *CC*, ser. LNCS, vol. 7120. Springer, 2012, pp. 144–164.

[11] P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, and A. Gupta, "Interprocedural exception analysis for C++," in *ECOOP*. Springer, 2011, pp. 583–608.

[12] G. Balakrishnan, N. Maeda, S. Sankaranarayanan, F. Ivančić, A. Gupta, and R. Pothengil, "Modeling and analyzing the interaction of C and C++ strings," in *FoVeOOS*, ser. LNCS, vol. 7421. Springer, 2011.

[13] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*. ACM, 2007, pp. 89–100.

[14] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*. Internet Society, 2008.

[15] W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Program analysis via satisfiability modulo path programs," in *POPL*. ACM, 2010, pp. 71–82.

[16] L. Jaulin, M. Kieffer, O. Didrit, and È. Walter, *Applied Interval Analysis*. London: Springer-Verlag, 2001.

[17] P. V. Hentenryck, D. McAllester, and D. Kapur, "Solving polynomial systems using a branch and prune approach," *SIAM J. on Numerical Analysis*, vol. 34, no. 2, pp. 797–827, 1997.

[18] P. Godefroid and J. Kinder, "Proving memory safety of floating-point computations by combining static and dynamic program analysis," in *ISSTA*. ACM, 2010.

[19] R. Majumdar, I. Saha, and Z. Wang, "Systematic testing for control applications," in *FMCAD*. IEEE, 2010, pp. 1–10.

[20] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, 2008.

[21] L. Granvilliers and F. Benhamou, "Algorithm 852: RealPaver: An interval solver using constraint satisfaction techniques," *ACM Trans. Math. Softw.*, vol. 32, no. 1, pp. 138–156, 2006.

[22] R. Sharma, M. Gligorić, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" in *FASE*. Springer, 2011.

[23] B. Dutertre and L. M. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 81–94.

[24] J. A. Clause and A. Orso, "LEAKPOINT: Pinpointing the causes of memory leaks," in *ICSE*. ACM, 2010, pp. 515–524.

[25] F. Ivančić, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki, "DC2: A framework for scalable, scope-bounded software verification," in *ASE*, 2011.

[26] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *ISSTA*. ACM, 2011.

[27] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *ICSE*. IEEE, 2012, pp. 727–737.

[28] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *CCS*, 2006.

[29] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA*. ACM, 2011, pp. 34–44.

[30] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE*. IEEE Comp. Society, 2007.

[31] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *TACAS*, 2005, pp. 365–381.

[32] J. Siddiqui and S. Khurshid, "ParSym: Parallel symbolic execution," in *ICSTE*. IEEE, 2010.

[33] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *EuroSys*. ACM, 2011, pp. 183–198.

[34] M. Kim, Y. Kim, and G. Rothermel, "A scalable distributed concolic testing approach: An empirical evaluation," in *ICST*. IEEE, 2012.