Mathematical Execution:
A Unified Approach for Testing Numerical Code

Zhoulai Fu
Zhendong Su
Department of Computer Science, University of California, Davis, USA
zhoulai.fu@gmail.com su@cs.ucdavis.edu

Abstract
This paper presents Mathematical Execution (ME), a new, unified approach for testing numerical code. The key idea is to (1) capture the desired testing objective via a representing function and (2) transform the automated testing problem to the minimization problem of the representing function. The minimization problem is to be solved via mathematical optimization. The main feature of ME is that it directly helps input space exploration by only executing the representing function, thus avoiding static or symbolic reasoning about the program semantics, which is particularly challenging for numerical code. To illustrate this feature, we develop an ME-based algorithm for coverage-based testing of numerical code. We also show the potential of applying and adapting ME to other related problems, including path reachability testing, boundary value analysis, and satisfiability checking.

To demonstrate ME’s practical benefits, we have implemented CoverMe, a proof-of-concept realization for branch coverage based testing, and evaluated it on Sun’s C math library (used in, for example, Android, Matlab, Java and JavaScript). We have compared CoverMe with random testing and Austin, a publicly available branch coverage based testing tool that supports numerical code (Austin combines symbolic execution and search-based heuristics). Our experimental results show that CoverMe achieves near-optimal and substantially higher coverage ratios than random testing on all tested programs, across all evaluated coverage metrics. Compared with Austin, CoverMe improves branch coverage from 43% to 91%, with significantly less time (6.9 vs. 6058.4 seconds on average).

1. Introduction
Testing has been a predominant approach for improving software quality. Manual testing is notoriously tedious [13]; automated testing has been an active research topic, drawing on a rich body of techniques, such as symbolic execution [12, 14, 17, 36], random testing [1, 11, 30] and search-based strategies [9, 38, 42, 49].

Automated testing is about producing program failures [51]. Let F00 be a program and dom(F00) be its input domain. An automated testing problem is to systematically find $x \in \text{dom}(F00)$ such that $F00(\mathbf{x}) \downarrow \text{wrong}$, where $F00(\mathbf{x}) \downarrow \text{wrong}$ denotes “F00 goes wrong if executed on input $\mathbf{x}$.” It is difficult to specify $F00(\mathbf{x}) \downarrow \text{wrong}$, which is known as the test oracle problem [50, 69]. This paper assumes that an algorithm for checking $F00(\mathbf{x}) \downarrow \text{wrong}$ is given.

An important problem in testing is the testing of numerical code, i.e., programs with floating-point arithmetic, non-linear variable relations, or external function calls (such as logarithmic and trigonometric functions). These programs are pervasive in safety-critical systems, but ensuring their quality remains difficult. Numerical code presents two specific challenges for existing automated testing techniques: (1) Random testing is easy to employ and fast, but ineffective in finding deep semantic issues and handling large input spaces; and (2) symbolic execution and its variants can perform systematic path exploration, but suffer from path explosion and are weak in dealing with complex program logic involving numerical constraints.

Our Approach. This paper introduces a new, unified approach for automatically testing numerical code. It proceeds as follows: We derive from the program under test $F00$ another program $F00_R$, called representing function, which represents how far an input $x \in \text{dom}(F00)$ is from reaching the set $\{x \mid F00(x) \downarrow \text{wrong}\}$. We require that the representing function returns a non-negative value for all $x$, which diminishes when $x$ gets close to the set and vanishes when $x$ goes inside. Intuitively, this representing function is similar to a sort of distance. It allows to approach the automated testing problem, i.e., the problem of finding an element in $\{x \mid F00(x) \downarrow \text{wrong}\}$, as the problem of minimizing $F00_R$. This approach can be justified with a strong guarantee:

$$F00(x) \downarrow \text{wrong} \Leftrightarrow x \text{ minimizes } F00_R,$$

assuming that there exists at least one $x$ such that $F00(x) \downarrow \text{wrong}$ (details in Sect. 4). Therefore, the essence of our approach is to transform the automated testing problem to a minimization problem. Minimization problems are well studied in the field of Mathematical Optimization (MO) [53]. MO works by executing its objective function only (see Sect. 2). That is to say, our approach does not need to analyze the semantics of the tested programs. Instead, it directly inputs space exploration by only executing the representing function.

We call this approach Mathematical Execution (abbreviated as ME).

Note that mathematical optimization by itself does not necessarily provide a panacea for automated testing because many MO problems are themselves intractable. However, efficient algorithms have been successfully applied to difficult mathematical optimization problems. A classic example is the NP-hard traveling salesman problem, which has been nicely handled by simulated annealing [37], a stochastic MO technique. Another example is Monte Carlo Markov Chain [8], which has been effectively adapted to testing and verification [28, 29, 34, 63]. A major finding of this work is that using mathematical optimization for testing numerical code is a powerful approach. If we carefully design the representing function so that certain conditions are respected, we can come up with mathematical optimization problems that can be efficiently solved by off-the-shelf MO tools.

To demonstrate the feasibility of our approach, we have applied ME on coverage-based testing [54] of floating-point code, a fundamental problem in testing. The experimental results show that our implemented tool, CoverMe, is highly effective. Fig. 1 gives a small program from our benchmark suite Fdlibm [3]. The program operates on two double input parameters. It first takes $|x|$’s high word by bit twiddling, including a bitwise AND (&), a pointer reference (&) and a dereference (*) operator. The bit twiddling result is stored in integer variable $i\mathbf{x}$ (Line 3), followed by four conditional statements that examine $i\mathbf{x}$ (Lines 4–15). The tool CoverMe yields:
#define __HI(x) *(1+(int)x&x)
double __kernel.cos(double x, double y){
  ix = __HI(x)&0x7fffffff; /* ix = |x|’s high word */
  if(ix<0x3e400000) { /* if |x| < 2**(-27) */
    if(((int)x)==0) return ...; /* generate inexact */
  }
  ...
  return ...;
}

Figure 1: The benchmark program __kernel.cos taken from the Fdlibm [3] library (http://www.netlib.org/fdlibm/k.cos.c).

100% line coverage
87.5% branch coverage

When investigating why CoverMe fails to achieve full branch coverage, we find that one out of the eight branches in the program cannot be reached. The condition if (((int)x) == 0) (Line 4) always holds because it is nested within the |x| < 2**(-27) branch (Line 4). Therefore, the 87.5% branch coverage is, in fact, optimal. We have compared CoverMe with Austin [43], a publicly available, state-of-the-art coverage-based testing tool that can handle floating-point code. Austin achieves 37.5% branch coverage in 1885.1 seconds, whereas CoverMe achieves the optimal coverage in 15.4 seconds (see Sect. 5).

Contributions. Our contributions follow:
• We introduce Mathematical Execution, a new general approach for testing numerical code;
• We develop an effective coverage-based testing algorithm using the ME approach;
• We demonstrate that ME is a unified approach by showing how to apply ME to several important testing problems; and
• We implement the coverage-based testing tool CoverMe and show its effectiveness on real-world numerical library code.

Paper Outline. Sect. 2 gives the background on mathematical optimization. Sect. 3 illustrates ME by studying the case of branch coverage based testing. We define the problem, demonstrate the ME solution, and give the algorithmic details. Sect. 4 lays out the theoretical foundation for ME and demonstrates ME with several additional examples. Sect. 5 presents an implementation overview of CoverMe and describes our experimental results. Sect. 6 discusses the current limitations of ME. Finally, Sect. 7 surveys related work and Sect. 8 concludes. For completeness, Appendix A lists the benchmark programs in Fdlibm that CoverMe does not support and their reasons, and Appendix B gives implementation details.

Notation. The sets of real and integer numbers are denoted by R and Z respectively. For two real numbers a and b, the usage aEb means a = 10b. In this presentation, we do not distinguish a mathematical expression, such as x^2 + |y|, and its implementa-

1Sun’s developers decided to use this redundant check to trigger the inexact exception of floating-point as a side effect. From the program semantics perspective, no input of __kernel.cos can trigger the false branch of if (((int)x) == 0).

2When investigating why CoverMe fails to achieve full branch coverage, we find that one out of the eight branches in the program cannot be reached. The condition if (((int)x) == 0) (Line 4) always holds because it is nested within the |x| < 2**(-27) branch (Line 4). Therefore, the 87.5% branch coverage is, in fact, optimal. We have compared CoverMe with Austin [43], a publicly available, state-of-the-art coverage-based testing tool that can handle floating-point code. Austin achieves 37.5% branch coverage in 1885.1 seconds, whereas CoverMe achieves the optimal coverage in 15.4 seconds (see Sect. 5).

2. Background

We begin with some preliminaries on mathematical optimization following the exposition of [28]. A complete treatment of either is beyond the scope of this paper. See [8, 53, 71] for more details.

A Mathematical Optimization (MO) problem is usually formulated as:

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad x \in S
\end{align*}
\]

where \(f\) is called the objective function, and \(S\) the search space. In general, mathematical optimization problems can be divided into two categories. One focuses on how functions are shaped at local regions and where a local minimum can be found near a given input. This local optimization is classic, usually involving standard techniques such as Newton’s or the steepest descent methods. Local optimization not only provides the minimum value of a function within a neighborhood of the given input points, but also aids global optimization, which determines the function minimum over the entire search space.

Local Optimization. Let \(f\) be a function defined over a Euclidean space with distance function \(d\). We call \(x^*\) a local minimum point if there exists a neighborhood \(x^*\), namely \(\{x \mid d(x,x') < \delta\}\) for some \(\delta > 0\), so that all \(x\) in the neighborhood satisfy \(f(x) \geq f(x^*)\). The value \(f(x^*)\) is called a local minimum of \(f\).

Local optimization problems can usually be efficiently solved if the objective function is smooth (such as continuous or differentiable to some degree) [56]. Fig. 2(a) shows a common local optimization method with the objective function \(\lambda x.x \leq 1 ? 0 : (x-1)^2\). It uses tangents of the curve to converge quickly to a minimum point. The smoothness of the curve makes it possible to deduce the function’s behavior in the neighborhood of a particular point \(x\) by using information at \(x\) only.

Global Optimization and MCMC. If \(f(x^*) \leq f(x)\) for all \(x\) in the search space, we call \(x^*\) a global minimum point (or minimum point for short), and \(f(x^*)\) the global minimum (or minimum for short) of the function \(f\). In this presentation, if we say “\(x\) minimizing the function \(f\)”, we mean \(x^*\) is a global minimum point of \(f\).

We use the Monte Carlo Markov Chain (MCMC) sampling to solve global optimization problems. A fundamental fact regarding
MCMC is that it follows the target distribution asymptotically. For simplicity, we give the results [8] with the discrete-valued probability.

**Lemma 2.1.** Let \( x \) be a random variable, \( A \) be an enumerable set of the possible values of \( x \). Let \( f \) be a target probability distribution for \( x \), i.e., the probability of \( x \) taking value \( a \) is \( f(a) \). Then, for an MCMC sampling sequence \( x_1, \ldots, x_n \) and a probability density function \( P(x_n = a) \) for each \( x_n \), we have \( P(x_n = a) \to f(a) \).

For example, consider the target distribution of coin tossing with 0.5 probability for having the head. An MCMC sampling is a sequence of random variables \( x_1, \ldots, x_n, \ldots \) such that the probability of \( x_n \) being "head", denoted by \( P_n \), converges to 0.5.

MCMC provides multiple advantages in practice. Because such sampling can simulate an arbitrary distribution (Lem. 2.1), MCMC backends can sample for a target distribution in the form of \( \lambda \cdot \exp(-f(x)) \) where \( f \) is the function to minimize, which allows its sampling process to attain the minimum points more frequently than the other points. Also, MCMC has many sophisticated techniques that integrate with classic local search techniques, such as the Basin-hopping algorithm [45] mentioned above. Some variants of MCMC can even handle high dimensional problems [60], or non-smooth objective functions [25]. Fig. 2(b) illustrates a typical MCMC cycle. Steps \( p_0 \to p_1, p_2 \to p_3, \) and \( p_4 \to p_5 \) are the local optimization; Steps \( p_1 \to p_2 \) and \( p_3 \to p_4 \) aim to prevent the MCMC sampling from getting trapped in the local minima.

## 3. Branch Coverage Based Testing

This section shows a detailed ME procedure in solving branch coverage based testing for numerical code.

### 3.1 Problem Statement

**Definition 3.1.** Let \( FOO \) be the program under test with \( N \) conditional statements, labeled by \( l_0, \ldots, l_{N-1} \). Each \( l_i \) has a true branch \( i_T \) and a false branch \( i_F \). The problem of branch coverage based testing aims to find a set of inputs \( X \subseteq \text{dom}(FOO) \) that covers all \( 2 \cdot N \) branches of \( FOO \). Here, we say a branch is “covered” by \( X \) if it is passed through by the path of executing \( FOO \) with an \( x \in X \). We scope the problem with three assumptions:

(a) The inputs of \( FOO \) are floating-point numbers.

(b) Each Boolean condition in \( FOO \) is an arithmetic comparison in the form of \( a \circ b \), where \( \circ \in \{ ==, \leq, <, \neq, >, \geq \} \), and \( a \) and \( b \) are floating-point variables or constants.

(c) Each branch of \( FOO \) is feasible, i.e., it is covered by \( \text{dom}(FOO) \).

Assumptions (a) and (b) are set for modeling numerical code. Assumption (c) is set to simplify our presentation. Our implementation will partially relax these assumptions (details in Appendix B).

We introduce the concept of a saturated branch and use it to reformulate Def. 3.1.

**Definition 3.2.** Let \( X \) be a set of inputs generated during the process of testing. We say a branch is saturated by \( X \) if the branch itself and all its descendant branches, if any, are covered by \( X \). Here, a branch \( b' \) is called a descendant branch of \( b \) if there exists a segment of control flow path from \( b \) to \( b' \). Given \( X \subseteq \text{dom}(FOO) \), we write

\[
\text{Saturate}(X) \tag{3}
\]

for the set of branches saturated by \( X \).

To illustrate Def. 3.2, suppose that an input set \( X \) covers \{0_F, 0_T, 1_F\} in the control-flow graph on the right, then Saturate(X) = \{0_F, 1_F\}; 1_T is not saturated because it is not covered, and 0_T is not saturated because its descendant branch 1_T is not covered.

![Image of a control-flow graph](image.png)

**Lemma 3.3.** Let \( FOO \) be a program under test with the assumptions Def. 3.1(a-c) satisfied. The goal of branch coverage based testing can be stated as to find a small set of inputs \( X \subseteq \text{dom}(FOO) \) that saturates all \( FOO \)’s branches.

**Remark 3.4.** In Lem. 3.3, we expect the generated input set \( X \) to be “small”, because otherwise we can use \( X = \text{dom}(FOO) \) which already saturates all branches under the assumption Def. 3.1(c).
3.2 An Example

We use a simple program F00 in Fig. 3 to illustrate our approach. The program has two conditional statements \( l_0 \) and \( l_1 \), and their true and false branches are denoted by \( 0 \) and \( 1 \) respectively. The objective is to find an input set that saturates all branches. Our approach proceeds in three steps:

Step 1. We inject a global variable \( r \) in F00, and, immediately before each control point \( l_i \), we inject an assignment (Fig. 3)

\[
\text{pen} = \lambda x.0
\]

where \( \text{pen} \) invokes a code segment with parameters specific to \( l_i \). The idea of \( \text{pen} \) is to capture the distance of the program input from saturating a branch that has not yet been saturated. As illustrated in Fig. 3, this distance returns different values depending on whether the branches at \( l_i \) are saturated or not.

We denote the instrumented program by F00. The key in Step 1 is to design \( \text{pen} \) to meet certain conditions that allow us to approach the problem defined in Lem. 3.3 as a mathematical optimization problem. We will specify the conditions in the next step.

Step 2. This step constructs the representing function that we have mentioned in Sect. 1. The representing function is the driver program F00.\( \text{R} \) shown in Fig. 3. It initializes \( r \) to 1, invokes \( \text{F00.} \) \( \text{I} \) and then returns \( r \) as the output of F00.\( \text{R} \). That is, \( \text{F00.} \text{R}(x) \) for a given input \( x \) calculates the value of \( r \) at the end of executing F00.\( \text{I}(x) \).

Our approach requires two conditions on F00.\( \text{R} \):

C1. F00.\( \text{R}(x) \geq 0 \) for all \( x \), and

C2. F00.\( \text{R}(x) = 0 \) if and only if \( x \) saturates a new branch. In other words, a branch that has not been saturated by the generated input set \( X \) becomes saturated with \( X \cup \{ x \} \), i.e., Saturate(\( X \) \( \neq \) Saturate(\( X \cup \{ x \} \)).

Imagine that we have designed \( \text{pen} \) so that F00.\( \text{R} \) meets both C1 and C2. Ideally, we can then saturate all branches of F00 by repeatedly minimizing F00.\( \text{R} \) as shown in the step below.

Step 3. In this step, we use MCMC to calculate the minimum points of F00.\( \text{R} \). Other mathematical optimization techniques, e.g., genetic programming [40], may also be applicable, which we leave for future investigation.

We start with an input set \( X = \emptyset \) and Saturate(\( X \)) = \( \emptyset \). We minimize F00.\( \text{R} \) and obtain a minimum point \( x^* \) which necessarily saturates a new branch by condition C2. Then we have \( X = \{ x^* \} \) and we minimize F00.\( \text{R} \) again which gives another input \( x^{**} \) and \( \{ x^*, x^{**} \} \) saturates a branch that is not saturated by \( \{ x^* \} \). We continue this process until all branches are saturated. When the algorithm terminates, F00.\( \text{R}(x) \) must be strictly positive for any input \( x \), due to C1 and C2.

Table 1: A scenario of how our approach saturates all branches of F00 by repeatedly minimizing F00.\( \text{R} \). Column “Saturate”. Branches that have been saturated. Column “\( \text{F00.}\text{R} \)”. The representing function and its plot. Column “\( x^* \)”. The point where F00.\( \text{R} \) attains the minimum. Column “\( X \)”: Generated test inputs.

| # | Saturate | F00.\( \text{R} \) | \( x^* \) | \( X \) |
|---|---|---|---|---|
| 1 | \( \emptyset \) | \( \lambda x.0 \) | 2 | \{0.7\} |
| 2 | \{1\} | \( \lambda x.((x+1)^2-4)^2 \) if \( x \leq 1 \) \( \text{else} \) | 1.0 | \{0.7, 1.0\} |
| 3 | \{0, 1\} | \( \lambda x.\left(\frac{(x-1)^2}{\varepsilon}\right) \text{if } x > 1 \) \( \text{else} \) | 1.1 | \{0.7, 1.0, 1.1\} |
| 4 | \{0, 1\} | \( \lambda x.1 \) | -5.2 | \{0.7, 1.0, 1.1, -5.2\} |

Tab. 1 illustrates a scenario of how our approach saturates all branches of F00. Each “\( \# \)” below corresponds to one line in the table. We use \( \text{pen}_0 \) and \( \text{pen}_1 \) to denote \( \text{pen} \) injected at \( l_0 \) and \( l_1 \) respectively. (\#1) Initially, Saturate(\( \emptyset \)) = \( \emptyset \). Any input injects a new branch. Both \( \text{pen}_0 \) and \( \text{pen}_1 \) set \( r = 0 \), and F00.\( \text{R} \) = \( \lambda x.0 \) (Fig. 3).

Suppose \( x^* = 0.7 \) is found as the minimum point. (\#2) The branch \( l_1 \) is now saturated and \( l_2 \) is not. Thus, \( \text{pen}_1 \) sets \( r = (y - 4)^2 \), Minimizing F00.\( \text{R} \) gives \( x^* = -3.0 \), 1.0, or 2.0. We have illustrated this MCMC procedure in Fig. 2(b). Suppose \( x^* = 1.0 \) is found. (\#3) Both \( l_1 \) and \( l_2 \), as well as \( l_0 \), are saturated by the generated inputs \{0.7, 1.0\}. Thus, \( \text{pen}_0 \) returns the previous \( r \). Then, F00.\( \text{R} \) amounts to \( \text{pen}_0 \), returning 0 \( \text{if } x > 1 \), or \((x - 1)^2 + \varepsilon \) otherwise, where \( \varepsilon \) is a small positive constant. Suppose \( x^* = 1.1 \) is found as the minimum point. (\#4) All branches have been saturated. In this case, both \( \text{pen}_0 \) and \( \text{pen}_1 \) return the previous \( r \). Then, F00.\( \text{R} \) becomes \( \lambda x.1 \), understanding that F00.\( \text{R} \) initializes \( r \) as 1. The minimum point, e.g.,

\( x^* = -5.2 \), necessarily satisfies F00.\( \text{R}(x^*) > 0 \), which terminates the algorithm.

Remark 3.5. Given an input \( x \), the value of F00.\( \text{R}(x) \) may change during the minimization process. In fact, F00.\( \text{R} \) is constructed with injected \( \text{pen} \) which returns different values at \( l_i \) depending on whether the branches \( l_i \) and \( l_f \) have been saturated. Thus, the minimization step in our algorithm differs from existing mathematical optimization techniques where the objective function is fixed [71].

3.3 Algorithm

We provide details corresponding to the three steps in Sect. 3.2. The algorithm is summarized in Alg. 1.

Algorithm for Step 1. The outcome of this step is the instrumented program F00.\( \text{I} \). As explained in Sect. 3.2, the essence is to inject the variable \( r \) and the assignment \( r = \text{pen} \) before each conditional statement (Algo. 1, Lines 1–4).

To define \( \text{pen} \), we first introduce a set of helper functions that are sometimes known as branch distance. There are many different forms of branch distance in the literature [38, 49]. We define ours with respect to an arithmetic condition \( a \ op \ b \).

Definition 3.6. Let \( a, b \in \mathbb{R}, \ op \in \{==, \leq, <, \neq, >, \geq\}, \varepsilon \in \mathbb{R}_{>0} \). We define branch distance \( d_e(a, b, \varepsilon) \) as follows:

\[
d_e(==, a, b) \triangleq (a - b)^2
d_e(\leq, a, b) \triangleq (a \leq b) \? 0 : (a - b)^2
d_e(<, a, b) \triangleq (a < b) \? 0 : (a - b)^2 + \varepsilon
d_e(\neq, a, b) \triangleq (a \neq b) \? 0 : \varepsilon
d_e(\geq, a, b) \triangleq d_e(\leq, b, a), d_e(>, a, b) \triangleq d_e(<, b, a)
\]

and \( d_e(a, b, \varepsilon) \triangleq d_e(\leq, b, a), d_e(a, b, \varepsilon) \triangleq d_e(<, b, a) \). Usually, the parameter \( \varepsilon \) is a small constant, so we drop the explicit reference to \( \varepsilon \) when using the branch distance.

The intention of \( d(a, b) \) is to quantify how far \( a \) and \( b \) are from attaining \( a \ op b \). For example, \( d(==, a, b) \) is strictly positive
when \( a \neq b \), becomes smaller when \( a \) and \( b \) go closer, and vanishes when \( a \approx b \). The following property holds:

\[
d(op, a, b) \geq 0 \quad \text{and} \quad d(op, a, b) = 0 \iff a \approx b.
\]  

(9)

As an analogue, we set \( pen \) to quantify how far an input is from saturating a new branch. We define \( pen \) following Algo. 1, Lines 14-23.

**Definition 3.7.** For branch coverage based testing, the function \( pen \) has four parameters, namely, the label of the conditional statement \( l_i \), and \( op, a, b \) and from the arithmetic condition \( a \approx b \).

(a) If neither of the two branches at \( l_i \) is saturated, we let \( pen \) return 0 because any input saturates a new branch (Lines 16-17).

(b) If one branch at \( l_i \) is saturated but the other is not, we set \( r \) to be the distance to the unsaturated branch (Lines 18-21).

(c) If both branches at \( l_i \) have already been saturated, \( pen \) returns the previous value of the global variable \( r \) (Lines 22-23).

For example, the two instances of \( pen \) at \( l_0 \) and \( l_1 \) are invoked as \( pen(l_0, \leq x, 1) \) and \( pen(l_1, \approx = y, 4) \) respectively in Fig. 3.

**Algorithm for Step 2.** This step constructs the representing function \( FOO_R \) (Algo. 1, Line 5). Its input domain is the same as that of \( FOO_I \) and \( FOO_R \), and its output domain is double, so to simulate a real-valued mathematical function which can then be processed by the mathematical optimization backend.

\( FOO_R \) initializes \( r \) to 1. This is essential in the correctness of the algorithm because we expect \( FOO_R \) returns a non-negative value when all branches are saturated (Sect. 3.2, Step 2). \( FOO_R \) then calls \( FOO_I \) and records the value of \( r \) at the end of executing \( FOO_I \). This \( r \) is the returned value of \( FOO_R \).

As mentioned in Sect. 3.2, it is important to ensure that \( FOO_R \) meets conditions C1 and C2. The condition C1 holds true since \( FOO_R \) returns the value of the instrumented \( r \), which is never assigned a negative quantity. The lemma below states \( FOO_R \) also satisfies C2.

**Lemma 3.8.** Let \( FOO_R \) be the program constructed in Algo. 1, and \( S \) the branches that have been saturated. Then, for any input \( x \) \in \text{dom}(FOO_R), FOO_R(x) = 0 \iff x \text{ saturates a branch that does not belong to } S.

**Proof.** We first prove the \( \Rightarrow \) direction. Take an arbitrary \( x \) such that \( FOO_R(x) = 0 \). Let \( \tau = [l_0, \ldots, l_m] \) be the path in \( FOO \) passed through by executing \( FOO(x) \). We know, from Lines 2-4 of the algorithm, that each \( l_i \) is preceded by an invocation of \( pen \) in \( FOO_R \). We write \( pen_i \) for the one injected before \( l_i \) and divide \( \{ pen_i \mid i \in [1, n] \} \) into three groups. For the given input \( x \), we let \( P1, P2 \) and \( P3 \) denote the groups of \( pen \) that are defined in Def. 3.7(a), (b) and (c), respectively. Then, we can always have a prefix path of \( \tau = [l_0, \ldots, l_m] \), with \( 0 \leq m \leq n \) such that each \( pen_i \) for \( i \in [m + 1, n] \) belongs to \( P3 \), and each \( pen_i \) for \( i \in [0, m] \) belongs to either \( P1 \) or \( P2 \). Here, we can guarantee the existence of such an \( m \) because, otherwise, all \( pen_i \) belong to \( P3 \), and \( FOO_R \) becomes \( \lambda x.1 \). The latter contradicts the assumption that \( FOO_R(x) = 0 \). Because each \( pen_i \) for \( i > m \) does nothing but performs \( r = r \), we know that \( FOO_R(x) \) equals to the exact value of \( r \) that \( pen_m \) assigns. Now consider two divisive cases on \( pen_m \). If \( pen_m \) is in \( P1 \), we immediately conclude that \( x \) saturates a new branch. Otherwise, if \( pen_m \) is in \( P2 \), we obtain the same from Eq. (9). Thus, we have established the \( \Rightarrow \) direction of the lemma.

To prove the \( \Leftarrow \) direction, we use the same notation as above, and let \( x \) be the input that saturates a new branch, and \( [l_0, \ldots, l_m] \) be the exercised path. Assume that \( M \) where \( 0 \leq m \leq n \) corresponds to the newly saturated branch. We know from the algorithm that (1) \( pen_m \) updates \( r \) to 0, and (2) each \( pen_i \) such that \( i > m \) maintains the value of \( r \) because their descendant branches have been saturated. We have thus proven the \( \Leftarrow \) direction of the lemma.

**Algorithm 1: Branch coverage based testing via ME.**

**Input:** \( FOO \quad \text{Program under test} \)

\( n_{\text{start}} \quad \text{Number of starting points} \)

\( LM \quad \text{Local optimization used in MCMC} \)

\( n_{\text{iter}} \quad \text{Number of iterations for MCMC} \)

**Output:** \( X \quad \text{Generated input set} \)

1. Inject global variable \( r \) in \( FOO \)

2. for conditional statement \( l_i \) in \( FOO \) do

3. Let the Boolean condition at \( l_i \) be a \( a \approx b \) where \( op \in \{\leq, <, =, >, \neq\} \)

4. Insert assignment \( r = pen(l_i, op, a, b) \) before \( l_i \)

5. Let \( FOO_I \) be the newly instrumented program, and \( FOO_R \) be:

\[
\text{double } FOO_R \text{ (double } x) \{ r = 1; FOO_I(x); return r; \}\]

6. Let \( \text{Saturate} = 0 \)

7. Let \( X = \emptyset \)

8. for \( k = 1 \) to \( n_{\text{start}} \) do

9. Randomly take a starting point \( x \)

10. Let \( x^* = \text{MCMC}(FOO_R, x) \)

11. if \( FOO_R(x^*) = 0 \) then \( X = X \cup \{x^*\} \)

12. Update Saturate

13. return \( X \)

**Function \( \text{pen}(l, op, a, b) \)**

14. Let \( \bar{x}_F \) and \( \bar{x}_L \) be the true and false branches at \( l_i \)

15. if \( \bar{x}_F \not\in \text{Saturate} \) and \( \bar{x}_L \not\in \text{Saturate} \) then

16. return 0

17. else if \( \bar{x}_F \not\in \text{Saturate} \) and \( \bar{x}_L \in \text{Saturate} \)

18. return \( d(op, a, b) / *: \text{Branch distance} \)

19. else if \( \bar{x}_F \in \text{Saturate} \) and \( \bar{x}_L \not\in \text{Saturate} \)

20. return \( d(\bar{x}_F, a, b) / *: \text{the opposite of } op \)

21. else if \( \bar{x}_F \in \text{Saturate} \) and \( \bar{x}_L \in \text{Saturate} \)

22. return \( r \)

**Function \( \text{MCMC}(f, x) \)**

23. \( x_L = \text{LM}(f, x) \)

24. for \( k = 1 \) to \( n_{\text{iter}} \) do

25. Let \( \delta \) be a random perturbation generation from a predefined distribution

26. Let \( x_L = \text{LM}(f, x_L + \delta) \)

27. if \( f(x_L) < f(x) \) then \( \text{accept} = \text{true} \)

28. else

29. Let \( m \) be a random number generated from the uniform distribution on \([0, 1]\)

30. let \( \text{accept} = \text{the Boolean } m < \exp(f(x_L) - f(x_L)) \)

31. if \( \text{accept} \) then \( x_L = x_L \)

32. return \( x_L \)

**Algorithm for Step 3.** The main loop (Algo. 1, Lines 8-12) relies on an existing MCMC engine. It takes an objective function and a starting point and outputs \( x^* \) that it regards as a minimum point. Each iteration of the loop launches MCMC from a randomly selected starting point (Line 9). From each starting point, MCMC computes the minimum point \( x^* \) (Line 10). If \( FOO_R(x^*) = 0 \), \( x^* \) is added to the set of the generated inputs \( X \) (Line 11). Lem. 3.8 ensures that \( x^* \) saturates a new branch in the case of \( FOO_R(x^*) = 0 \). Therefore, in theory, we only need to set \( n_{\text{start}} = 2 + N \) where \( N \) denotes the number of conditional statements, so to saturate all \( 2 + N \) branches. In practice, however, we set \( n_{\text{start}} = 2 + N \) because MCMC cannot guarantee that its output is a true global minimum point.
The MCMC procedure (Algo. 1, Lines 24-34) is also known as the Basin hopping algorithm [45]. It is an MCMC sampling over the space of the local minimum points [46]. The random starting point \( x \) is first updated to a local minimum point \( x_1 \) (Line 25). Each iteration (Lines 26-33) is composed of the two phases that are classic in the Metropolis-Hastings algorithm family of MCMC [16]. In the first phase (Lines 27-28), the algorithm proposes a sample \( \tilde{x}_1 \) from the current sample \( x \). The sample \( \tilde{x}_1 \) is obtained with a perturbation \( \delta \) followed by a local minimization, \( \tilde{x}_1 = Lm(f, x_1 + \delta) \) (Line 28), where \( Lm \) denotes a local minimization in Basin hopping, and \( f \) is the objective function. The second phase (Lines 29-33) decides whether the proposed \( \tilde{x}_1 \) should be accepted as the next sampling point. If \( f(\tilde{x}_1) < f(x_1) \), the proposed \( \tilde{x}_1 \) will be sampled; otherwise, \( \tilde{x}_1 \) may still be sampled, but only with the probability of \( \exp((f(x_1) - f(\tilde{x}_1))/T) \), in which \( T \) (called the annealing temperature [37]) is set to 1 in Algo. 1 for simplicity.

4. Mathematical Execution

Predicate as function is a common concept in mathematics. As Gödel stated in his 1931 work [22, 31],

There shall correspond to each relation \( R \) \( R \subseteq \mathbb{Z}^n \) a representing function \( \phi(x_1, \ldots, x_n) = 0 \) if \( R(x_1, \ldots, x_n) \) and \( \phi(x_1, \ldots, x_n) = 1 \) if \( \neg R(x_1, \ldots, x_n) \).

Traditional representing function in its Boolean nature is a predicate/set indicator of two states, essentially being true or false. For example, \( \text{even}(N) \) that decides whether an integer \( N \) is even can be represented by the function \( \lambda x(x \mod 2 = 0) \).

In this section, we present Mathematical Execution (ME) by extending the Boolean-valued representing function to a real-valued function, which allows us to approach a search problem by minimizing its representing function.

4.1 Search Problem

Definition 4.1. The search problem with regard to a set \( X \) aims to

(a) find an \( x \in X \) if \( X \neq \emptyset \), and
(b) report “not found” if \( X = \emptyset \).

Usually, we have a search space \( U \), and \( X \) is specified implicitly as a subset of \( U \). We denote the search problem by \((X, U)\). In this paper, we deal with numerical code, and thus, we assume that \( \forall x \in X \), \( x \) is a real number.

Example 4.2. A search problem can be any computational task that attempts to find an element from a set.

(a) As per the notation used in Sect. 1, an automated testing program of program \( F_0 \) is a search problem \((X, U)\) where \( X = \{ x \mid F_0(x) \uparrow \text{ wrong} \} \) and \( U = \text{dom}(F_0) \).

(b) Another search problem is satisfiability checking, where \( X \) is the set of the models of a constraint, and \( U \) is the value domain to which the variables can be assigned.

4.2 Representing Function

Definition 4.3. A function \( R \) is said to be a representing function for the search problem \((X, U)\) if with any \( x \in U \) there is an associated real value \( R(x) \), such that

(a) \( R(x) \geq 0 \);
(b) every root of the representing function is a solution of the search problem, \( \text{i.e., } R(x) = 0 \implies x \in X \); and
(c) the roots of \( R \) include all solutions to the search problem, \( \text{i.e., } x \in X \implies R(x) = 0 \).

Example 4.4. Let \((X, U)\) be a search problem.

(a) A trivial representing function is \( \lambda x.(x \in X)? 0: 1 \).
(b) A generic representing function is the point-set distance. Imagine that the search problem is embedded in a metric space [62] with a distance \( \text{dist} : X \times X \to \mathbb{R} \). As a standard practice, we can lift \( \text{dist} \) to \( \text{dist}_X \) defined as \( \lambda x.\inf\{\text{dist}(x, x') \mid x' \in X\} \), where inf refers to the greatest lower bound, or infimum. Intuitively, \( \text{dist}_X \) measures the distance between a point \( x \in U \) and the set \( X \). It can be shown that \( \text{dist}_X \) satisfies conditions Def. 4.3(a-c), and therefore, is a representing function.

(c) The representing function used in branch coverage based testing is the FOO_R constructed in Sect. 3, where \( X \) is the input that saturates a new branch and \( U \) is the input domain. We have proved that FOO_R is a representing function in Lem. 3.8.

The theorem below allows us to approach a search problem by minimizing its representing function.

Theorem 4.5. Let \( R \) be the representing function for the search problem \((X, U)\), and \( R^* \) be the global minimum of \( R \).

(a) Deciding the emptiness of \( X \) is equivalent to checking the sign of \( R^* \), i.e., \( X = \emptyset \Leftrightarrow R^* > 0 \).

(b) Assume \( X \neq \emptyset \). Then, \( \forall x \in U, x \) minimizes \( R \Leftrightarrow x \in X \).

Proof. Suppose \( x \neq \emptyset \). Let \( x_0 \) be an element of \( X \). We have \( R^\ast \geq 0 \) by Def. 4.3(a). In addition, we have \( R^\ast \leq R(x_0) \) since \( R^\ast \) is the minimum. Then we have \( R^\ast \leq 0 \) because \( R(x_0) = 0 \) due to Def. 4.3(c). Thus \( R^\ast = 0 \). Conversely, \( R^\ast = 0 \) implies that there exists an \( x^\ast \in U \text{ s.t. } R(x^\ast) = 0 \). By Def. 4.3(b), \( x^\ast \in X \). Thus \( X = \emptyset \).

Proof of (b): Let \( 0_R \) denote the set of the roots of \( R \), and \( M_R \) be the set of the minimum points of \( R \). It suffices to show that \( 0_R \subseteq M_R \subseteq X \subseteq 0_R \). We have \( 0_R \subseteq M_R \) because a root of \( R \) is necessarily a minimum point; we have \( M_R \subseteq X \) because \( x \neq \emptyset \) implies \( R^\ast = 0 \) by Thm. 4.5(a). Take an arbitrary \( x^\ast \in M_R, R(x^\ast) = R^\ast = 0 \), and therefore \( x^\ast \in X \) by Def. 4.3(b); we have \( X \subseteq 0_R \) from Def. 4.3(c).

4.3 The ME procedure

The representing function paves the way toward a generic solution to the search problem. The key parameter in the ME procedure is mathematical optimization.

Definition 4.7. Let \( U \) be a set, \( \mu \) be a mathematical optimization algorithm that attempts to calculate a minimum point for an objective function defined over \( U \). We define the Mathematical Execution procedure as follows:

Input: A search problem \((X, U)\)
Output: An element of \( X \), or “not found”

M1. Construct the representing function \( R \).
M2. Minimize \( R \). Let \( x^\ast \) be the minimum point obtained by \( \mu \).
M3. Return \( x^\ast \) if \( x^\ast \in X \). Otherwise return “not found”.

The following corollary states that the ME procedure solves the search problem, under the condition that the ME procedure is equipped with an ideal mathematical optimization backend. Its proof (omitted) follows Thm. 4.5.
Corollary 4.8. Let \((X, U)\) be a search problem. Assume that \(\mu\) yields a true global minimum point in \(M_2\) of the ME procedure. Then,

(a) the ME procedure returns an \(x^* \in X\) if \(X \neq \emptyset\), and

(b) the ME procedure returns “not found” if \(X = \emptyset\).

Remark 4.9. A known issue with testing is its incompleteness, i.e., “Testing shows the presence of bugs, but not its absence” [24]. In the context of Mathematical Execution, the above well-known remark corresponds to the fact that, in practice, if the ME procedure returns “not found” for the search problem \((X, U)\), \(X\) may still be non-empty. We call this phenomenon practical incompleteness.

The practical incompleteness occurs when the MO backend fails to yield an accurate minimum point. To clarify, we use \(x^*\) for an exact global minimum point, and we use \(x^t\) for the one calculated by the MO backend \(\mu\). Then we consider four disjunctive cases:

(a) \(R(x^*) = 0\) and \(R(x^t) = 0\);
(b) \(R(x^*) > 0\) and \(R(x^t) > 0\);
(c) \(R(x^*) = 0\) and \(R(x^t) > 0\);
(d) \(R(x^*) > 0\) and \(R(x^t) = 0\).

The ME procedure remains correct for both (a) and (b). The case (c) cannot happen because \(R(x^*) < R(x)\) for all \(x\). The practical incompleteness occurs in (d), where the ME procedure returns “not found” but \(X \neq \emptyset\). Sect. 6 further discusses this incompleteness.

4.4 Additional Examples

This subsection aims to show that ME is a unified approach by applying it on several other important search problems besides coverage-based testing. In each example, we illustrate ME with a different representing function.

4.4.1 Path Reachability Testing

Given a path \(\tau\) of program \(\text{F00}\), we call path reachability testing the search problem \((X, U)\) with

\[
X = \{x \mid x \text{ triggers the path } \tau\}, U = \text{dom}(\text{F00}).
\]  

(10)

The path reachability problem has been studied as an independent research topic [52], or more commonly, as a subproblem in other testing problems [30, 38, 42].

Consider the program \(\text{F00}\) in Fig. 4(left), which is the same as that in Fig. 3. Suppose that we want to trigger the path \(\tau = [0_{\tau}, 1_{\tau}]\) (we denote a path by a sequence of branches). Our approach to this example problem is similar to the three steps explained in Sect. 3.2, except that we design a different representing function here. We illustrate the ME approach in Fig. 4.

Step 1. We inject a global variable \(r\) in \(\text{F00}\) and the assignment

\[
r = r + d(op, a, b)
\]  

(11)

before the conditional statements, where \(d\) is the branch distance defined in Def. 3.6. The instrumented program is shown as \(\text{F00} \_I\) in Fig. 4. The assignment is to measure how far the input has attained the desired path.

Step 2. The value of \(r\) is then retrieved through a driver program \(\text{F00} \_R\) (Fig. 4), which initializes \(r\) to 0 (unlike in Sect. 3.2, where \(r\) is initialized to 1), calls \(\text{F00} \_I\) and then returns \(r\).

Step 3. A global minimum point of \(\text{F00} \_R\) is calculated by an MO algorithm. As shown in the graph of \(\text{F00} \_R\) (Fig. 4), there are three local minimum points, \([-3, 1, 2]\). Two of them, \([-3, 1]\), attain the global minimum 0 and either solves the path reachability testing problem. Note that the representing function is discontinuous at \(x = 1\), but the MCMC procedure can easily find it (similar to Fig. 2(b)).

Below we prove the correctness of the ME solution.

Corollary 4.10. An input \(x\) triggers \(\tau\) iff \(x\) minimizes \(\text{F00} \_R\).

Proof. First, it can be shown that the constructed \(\text{F00} \_R\) is a representing function for the path reachability problem, thus we can apply Thm. 4.5. By Thm. 4.5(a), \(X \neq \emptyset\) since \(\text{F00} \_R\) attains the global minimum 0. Then we conclude from Thm. 4.5(b). ∎

4.4.2 Boundary Value Analysis

In testing, test inputs that explore “boundary conditions” usually have a higher payoff than those that do not [70]. The problem of generating such inputs is expressed abstractly as boundary value analysis [39, 54, 58], which can be seen as the search problem with

\[
X = \{x \mid x \text{ triggers a boundary condition}\}, U = \text{dom}(\text{F00}).
\]  

(12)

Consider again the program \(\text{F00}\) in Fig. 4. The boundary value analysis is to find test inputs that trigger a boundary condition (a) \(x = 1\) at \(l_0\), or (b) \(y = 4\) at \(l_1\). With manual reasoning, condition (a) can only be triggered by input 1. Condition (b) can be triggered if \(x = 2\) or \(x = -2\) before \(l_1\). For each case, we reason backward across the two branches at \(l_0\) and then merge the results. Eventually we have \([-3, 1, 2]\) as the test inputs to generate.
Our ME solution follows: As before, we introduce a global variable \( r \) to estimate how far a program input is from triggering a boundary condition. We inject the assignment

\[
r = r + d(==, a, b)
\]

before each condition \( a \ op \ b \), where function \( d \) is the branch distance defined in Def. 3.6. Fig. 5 illustrates FOO_I and FOO_R. Then we can solve the boundary value analysis problem via minimizing FOO_R. The correctness of this procedure follows:

**Corollary 4.11.** An input \( x \) triggers a boundary condition if and only if \( x \) minimizes FOO_R.

*Proof.* It can be shown that (1) \( r \) is always assigned to a non-negative value; (2) if \( r = 0 \) at the end of the program execution, then one of the boundary conditions must hold; and (3) \( r = 0 \) at the end of the program execution if at least one of the boundary conditions holds. Thus, the constructed FOO_R satisfies the conditions for being a representing function (Def. 4.3). We can then prove the corollary following Thm. 4.5 (in the same way as we prove Cor. 4.10).

### 4.4.3 Satisfiability Checking

Consider the satisfiability problem with constraint

\[
\pi = 2^x \leq 5 \land x^2 \geq 5 \land x \geq 0
\]

where \( x \in \mathbb{R} \). If we write \( x^* \models \pi \) to mean that \( \pi \) becomes a tautology by substituting its free variable \( x \) with \( x^* \). Then, this satisfiability problem can be expressed as the search problem \( \{ (x^* \mid x^* \models \pi), \mathbb{R} \} \). Its representing function can be defined as (Fig. 6):

\[
R \overset{\text{def}}{=} \lambda x. (2^x \leq 5) \land (x^2 \geq 5) \land (x \geq 0) \land (x^2 \geq 5) \land (x^2 \geq 5)
\]

Then we can solve the satisfiability checking problem of constraint \( \pi \) in Eq. (14) by minimizing its representing function \( R \) in Eq. (15). The ME procedure can easily locate the minimum points in between \( \sqrt{2} \approx 2.44 \) and \( \log_2 5 \approx 2.32 \). Each of the minimum points is necessarily a model of \( \pi \) following Thm. 4.5. The correctness of such an approach follows:

**Corollary 4.12.** \( x^* \models \pi \iff x^* \text{ minimizes } R \).

**Remark 4.13.** The representing function in Eq. (15) is defined on \( \mathbb{R} \). It is to illustrate the concept and allows us to ignore issues about floating-point inaccuracy. A more realistic representing function for satisfiability checking has been shown in a recent work [28] where the representing function is constructed based on the binary forms of floating-point numbers to avoid floating-point arithmetic in the first place.

### 5. Evaluation

#### 5.1 The ME-powered System CoverMe

We have implemented CoverMe, a proof-of-concept realization for branch coverage based testing. CoverMe has a modular design and can be adapted to other automated testing problems, such as path reachability and boundary value analysis (Sect. 4.4). This section presents the architecture of CoverMe (Fig. 7), in which we identify three layers and their corresponding use cases. Implementation details are given in Appendix B.

**Client.** This layer provides the program under test F00 and specifies the inputs to generate \( X \). The search problem involved will be \( (X, \text{dom}(F00)) \). \( X \) is usually implicitly, e.g., specified via a path to trigger as in path reachability testing (Sect. 4.4.1), and checking \( x \in X \) has to be feasible following Def. 4.1. F00 is a piece of code in the LLVM intermediate representation [44] or in any language that can be transformed to it (such as code in Ada, the C/C++ language family, or Julia). The current CoverMe has been tested on C, and we require \( \text{dom}(F00) \subseteq \mathbb{R}^N \), as the system backend outputs floating-point test data by default (see the ME kernel layer below).

**Researcher.** This layer sets two parameters. One is the initial value of the representing function, denoted by \( r_0 \). It is usually either 0 or 1 from our experience. For example, \( r_0 \) is set to 0 in path reachability testing (Sect. 4.4.1), and 1 in both coverage-based testing (Sect. 3.2) and boundary value analysis (Sect. 4.4.2). The other parameter is the assignment to inject before each conditional statement in F00. In practice, the Researcher specifies this code segment in the pen procedure and injects \( r = \text{pen} \), as shown in Sect. 3.

**ME Kernel.** This layer takes as inputs the program F00 provided by the Client, \( r_0 \) and pen set by the Researcher, and operates the three steps described in Sect. 3.2. It (1) uses Clang [2] to inject \( r \) and pen into F00 to construct F00_I, (2) initializes \( r \) to \( r_0 \), invokes F00_I and returns \( r \) in F00_R, and (3) minimizes F00_R with MO. As mentioned in Sect. 3.3, CoverMe uses Basinhopping, an off-the-shelf implementation from SciPy optimization package [7] as the MO backend. Basinhopping is then launched from different starting points as shown in Algo. 1, Lines 8-12. These starting points are randomly generated from the Hypothesis library [5].

#### 5.2 Experimental Setup

This section discusses the benchmarks, the tools for comparison, the settings and hardware, and two evaluation objectives.
Benchmarks. We use the C math library Fdlibm 5.3 [3] as our benchmark. These programs are developed by Sun (now Oracle). They are real-world programs, rich in floating-point operations, and have been used in Matlab, Java, JavaScript, and Android.

Fdlibm includes 80 programs. Each has one or multiple entry functions. In total, Fdlibm has 92 entry functions. Among them, we exclude (1) 36 functions that do not have branches, (2) 11 functions involving non-floating-point input parameters, and (3) 5 static C functions. Our benchmark suite includes all remaining 40 functions in Fdlibm. For completeness, we list untested functions and reasons why they are not selected in Appendix A.

Compared Tools. We have considered tools publicly available to us, including Austin [43], Pex [68], and Klee [15] and its two variants, namely Klee-Multisolver [57] and Klee-FP [18]. We tried Klee-Multisolver [57] with Z3 [23] as the SMT backend but found that the expression language of Klee [15] did not support floating-point constraints. Besides, some common operations in our benchmark programs, such as pointer reference and dereference, type casting, external function calls, etc., are not supported by Z3 or any other backend solvers compatible with Klee-Multisolver. Klee-FP supports symbolic reasoning on the equivalence between floating-point values but does not support coverage testing [18]. Pex, unfortunately, can only run for .NET programs on Windows whereas Fdlibm is in C, and our testing platform is based on Unix.

To our best knowledge, Austin [43] is the only publicly available tool supporting coverage testing for Fdlibm. Austin combines symbolic execution and search-based heuristics, and has been thoroughly tested for branch coverage based testing on floating-point programs [41]. For empirical comparison, we have also implemented a random sampling tool, which samples inputs from the function’s input domains using a standard pseudo-random number generator. We refer to the tool as Rand.

Settings and Hardware. CoverMe supports the following command line options while generating test inputs from floating-point programs: (1) the number of Monte-Carlo iterations n_iter, (2) the local optimization algorithm LM and (3) the number of starting points n_start. These options correspond to the three input parameters in Algo. 1. We set n_iter = 5, n_start = 500, and LM=“powell” which refers to Powell’s local optimization algorithm [59].

For both Austin and CoverMe, we use the default settings for running the benchmarks. All experiments were performed on a laptop with a 2.6 GHz Intel Core i7 and 4GB RAM running Ubuntu 14.04 virtual machine.

Evaluation Objectives. There are two specific evaluation objectives. (1) Coverage: We use the standard GNU coverage tool Gcov [4] to analyze the coverage. Gcov generates four metrics for source code coverage analysis, which are listed as “Lines”, “Conditions”, “Branches” and “Calls” in Col. 1, Tab. 2. Col. 2-4 give the corresponding Gcov report message, the descriptions and the general metrics for coverage testing. (2) Efficiency: we measure the wall time reported by the standard Unix command “time”. The timeout limit for the compared tools is 48 hours.

5.3 Quantitative Results

This subsection presents two sets of experimental results. The first validates our approach by comparing it against random testing. The second compares CoverMe with Austin [43], an open-source tool that combines symbolic execution and search-based strategies.

5.3.1 CoverMe versus Random Testing

We have compared CoverMe with Rand by running them on the benchmarks described in Sect. 5.2. In Tab. 3, we sort all benchmark programs (Col. 1) and entry functions (Col. 2) by their names, give the numbers of source lines, branches and invoked functions (Col. 3-5). All coverage results are given by Gcov. It can be seen that the programs are rich in branches. The largest number of branches is 114 (ieee754_atan2). Especially, some tested functions have more branches than lines. For example, ieee754_atan2 has 44 branches, but only 39 lines; ceil has 30 branches, but only 29 lines.

The times used by CoverMe are given by Col. 6 in Tab. 3. Observe that the times used by CoverMe vary considerably through all entry functions, from 0.1 to 22.1 seconds, but all under half a minute. We find the numbers of lines (Col. 3) and branches (Col. 4) are not correlated with the running times (Col. 6). CoverMe takes 1.1 seconds to run the function exp2 (with 42 branches and 56 lines) and 10.1 seconds to run the function floor (with 30 branches and 30 lines). It shows the potential for real-world program testing since CoverMe may not be very sensitive to the number of lines or branches. We set the timeout limit of Rand as 600 seconds, since Rand does not terminate by itself and 600 seconds are already larger than the times spent by CoverMe by orders of magnitude.

Col. 7-14 in Tab. 3 show the coverage results of Rand and CoverMe. We use Gcov to compute four metrics: lines, condition, branches and calls (as defined in Tab. 2). All coverage results reported by Gcov have been included in Tab. 3 for completeness. The columns “Line” and “Branch” refer to the commonly used line/statement coverage and branch coverage, respectively. The average values of the coverage are shown in the last row of the table. All values in Col. 8, 10, 12 and 14 are larger than or equal to the corresponding values in Col. 7, 9, 11 and 13. It means that CoverMe achieves higher coverage than Rand for every benchmark program and every metric. For the line and condition coverages, CoverMe achieves full coverage for almost all benchmarks, with 97.0% line coverage and 98.2% condition coverage on average, whereas Rand achieves 54.2% and 61.2% for the two metrics. For the most important branch coverage (since it is a primary target of this paper), CoverMe achieves 100% coverage for 11 out of 40 benchmarks with an average of 90.8% coverage, while Rand does...
not achieve any 100% coverage and attains only 38.0% coverage on average. CoverMe achieves 100% call coverage on average, whereas Rand achieves 39.5%.

### 5.3.2 CoverMe versus Austin

Table 4 reports the testing results of Austin and CoverMe. It uses the same set of benchmarks as Tab. 3 (Col. 1-2). We use the time (Col. 3-4) and the branch coverage metric (Col. 5-6) to evaluate the efficiency and the coverage. We choose branch coverage instead of the other three metrics in Gcov since branch coverage is the major concern in this paper. Besides, Gcov needs to have access to the generated test inputs to report the coverage, but currently, there is no viable way to access the test inputs generated by Austin. Unlike Rand, the branch coverage percentage of Austin (Col. 5) is provided by Austin itself, rather than by CoverMe.

Austin also shows large performance variances over different benchmarks, from 667.1 seconds (the program $\sin$) to hours. As shown in the last row of Tab. 4, Austin needs 6058.4 seconds on average for the testing. It should be mentioned that the average time does not include the benchmarks where Austin crashes or times out. Compared with Austin, CoverMe is faster (Tab. 4, Col. 4) with 6.9 seconds on average (The results are also shown in Tab. 3(Col. 6)). CoverMe achieves a much higher branch coverage (90.8%) than Austin (42.8%). We also compare across Tab. 4 and Tab. 3. On average, Austin provides slightly better branch coverage (42.8%) than Rand (38.0%).

Col. 7-8 are the improvement metrics of CoverMe against Austin. The Speedup (Col. 7) is calculated as the ratio of the time spent by Austin and the time spent by CoverMe. The coverage improvement (Col. 8) is calculated as the difference between the branch coverage of CoverMe and that of Austin. We observe that CoverMe provides 3,868X speedup and 48.9% coverage improvement on average.

**Remark 5.1.** Our evaluation shows that CoverMe has achieved high code coverage on most tested programs. One may wonder...
whether the generated inputs have triggered any latent bugs. Note that when no specifications are given, program crashes have been frequently used as an oracle for finding bugs in integer programs. Floating-point programs, on the other hand, can silently produce wrong results without crashing. Thus, program crashes cannot be used as a simple, readily available oracle as for integer programs. Our experiments, therefore, have focused on assessing the efficiency of ME in solving the problem defined in Def. 3.1 or Lem. 3.3, and do not evaluate its effectiveness in finding bugs, which is orthogonal and interesting future work.

6. Incompleteness

The development of ME toward a general solution to the search problem, as illustrated in this presentation, has been grounded in the concept of representing function and mathematical optimization. While our theory guarantees that the ME procedure solves the search problem correctly (Cor. 4.8), the practical incompleteness (Remark 4.9) remains a challenge in applying ME to real-world automated testing problems. Below we discuss three sources of practical incompleteness and how it can be mitigated.

Well-behaved Representing Functions. The well-behavedness of the representing function is essential for the working of ME. Well-behavedness is a common term in the MO literature to describe that when no specifications are given, program crashes have been frequently used as an oracle for finding bugs in integer programs. Floating-point programs, on the other hand, can silently produce wrong results without crashing. Thus, program crashes cannot be used as a simple, readily available oracle as for integer programs. Our experiments, therefore, have focused on assessing the efficiency of ME in solving the problem defined in Def. 3.1 or Lem. 3.3, and do not evaluate its effectiveness in finding bugs, which is orthogonal and interesting future work.

Table 4: Comparison of Austin [43] and CoverMe. The benchmark programs are taken from the Fdlibm library [3].

| Benchmark | Time (second) | Branch coverage(%) | Improvement metrics |
|-----------|---------------|--------------------|---------------------|
| e_acos.c  | 6058.8        | 7.8                | 16.7                |
| e_acosh.c | 201.6         | 2.3                | 90.0                |
| e_asin.c  | 6935.6        | 8.0                | 14.3                |
| e_atan2.c | 14456.0       | 17.4               | 34.1                |
| e_atanh.c | 4033.8        | 8.1                | 8.3                 |
| e_cosh.c  | 27334.5       | 8.2                | 37.5                |
| e_exp.c   | 2952.1        | 8.4                | 75.0                |
| e_fmod.c  | 5873.9        | 9.0                | 53.3                |
| e_hypot.c | 5456.8        | 15.6               | 36.4                |
| e_j0.c    | 6973.0        | 9.0                | 33.3                |
| e_j1.c    | 5838.3        | 0.7                | 56.3                |
| e_kos.c   | 5701.7        | 0.7                | 56.3                |
| e_log.c   | 5109.0        | 3.4                | 59.1                |
| e_log10.c | 1175.5        | 1.1                | 62.5                |
| e_pow.c   | 1175.5        | 1.1                | 62.5                |
| e_rem_pio2.c | 6429.0   | 2.2                | 45.5                |
| e_remainder.c | 1989.8     | 8.5                | 57.1                |
| e_scalb.c | 5534.8        | 0.6                | 35.0                |
| e_sinh.c  | 1431.6        | 10.2               | 50.0                |
| e_sqrt.c  | 5701.7        | 0.7                | 56.3                |
| e_tan.c   | 5109.0        | 3.4                | 59.1                |
| s_asinh.c | 2439.1        | 8.4                | 41.7                |
| s_atan.c  | 7584.7        | 8.5                | 26.9                |
| s_cbrt.c  | 3583.4        | 0.4                | 50.0                |
| s_cell.c  | 7166.3        | 8.8                | 36.7                |
| s_cos.c   | 669.4         | 0.4                | 75.0                |
| s_erf.c   | 20419.3       | 9.0                | 30.0                |
| s_exp1.c  | 6611.8        | 0.1                | 25.0                |
| s_floor.c | 7620.6        | 10.1               | 36.7                |
| s_ilogb.c | 3654.7        | 8.3                | 16.7                |
| s_log1p.c | 11913.7       | 9.9                | 61.1                |
| s_logb.c  | 1064.4        | 0.3                | 50.0                |
| s_mod.c   | 1795.1        | 3.5                | 50.0                |
| s_nextafter.c | 7777.3    | 17.5               | 50.0                |
| s_rint.c  | 5355.8        | 3.0                | 35.0                |
| s_sin.c   | 667.1         | 0.3                | 75.0                |
| s_tan.c   | 704.2         | 0.3                | 50.0                |
| s_tanh.c  | 2805.5        | 0.7                | 33.3                |
| MEAN      | 6058.4        | 6.9                | 42.8                |

Table 4: Comparison of Austin [43] and CoverMe. The benchmark programs are taken from the Fdlibm library [3].
null
References

[1] American fuzzy lop. http://clamtf.coredump.cx/afl/. Accessed: 25 June, 2016.

[2] Clang: A C language family frontend for LLVM. http://clang.llvm.org/. Accessed: 25 June 2016.

[3] Freely distributed math library, FDLIBM. http://www.netlib.org/fdlibm/. Accessed: 25 June 2016.

[4] GNU compiler collection tool, Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html/. Accessed: 25 June 2016.

[5] Python testing library, hypothesis. https://github.com/DRMacIver/hypothesis. Accessed: 25 June 2016.

[6] LLVM: Pass class reference. http://llvm.org/docs/doxygen/html/classllvm_1_1Pass.html. Accessed: 25 June 2016.

[7] Scipy optimization package. http://docs.scipy.org/doc/scipy-dev/reference/optimize.html. Accessed: 25 June 2016.

[8] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. Machine Learning, 50(1-2):43–50, 2003.

[9] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE ’11, pages 53–62, Washington, DC, USA, 2011.

[10] D. H. Bailey. High-precision floating-point arithmetic in scientific computation. Computing in Science and Engng., 7(3):54–61, May 2005.

[11] D. L. Bird and C. U. Muñoz. Automatic generation of random self-checking test cases. IBM Syst. J., 22(3):229–245, Sept. 1983.

[12] R. S. Boyer, B. Elspas, and K. N. Levitt. A formal system for testing and debugging programs by symbolic execution. In Proceedings of the International Conference on Reliable Software, pages 234–245, New York, NY, USA, 1975.

[13] F. P. Brooks, Jr. The Mythical Man-month (Anniversary Ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. Commun. ACM, 56(2):82–90, 2013.

[15] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI’08, pages 209–224, Berkeley, CA, USA, 2008.

[16] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. The American Statistician, 49(4):327–335, Nov. 1995.

[17] L. A. Clarke. A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng., 2(3):215–222, May 1976.

[18] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In Proceedings of the sixth conference on Computer systems, pages 315–328, 2011.

[19] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI’08, pages 209–224, Berkeley, CA, USA, 2008.

[20] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL’77, pages 238–252, 1977.

[21] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL’79, pages 269–282, New York, NY, USA, 1979.

[22] M. Davis. The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions. Dover Publications, Incorporated, 2004.

[23] L. De Moura and N. Björner. Z3: An efficient SMT solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’08/ETAPS’08, pages 337–340, Berlin, Heidelberg, 2008.

[24] E. W. Dijkstra. Notes on structured programming. Apr. 1970.

[25] J. Eckstein and D. P. Bertsekas. On the Douglas—Rachford splitting method and the proximal point algorithm for maximal monotone operators. Mathematical Programming, 55(1-3):293–318, 1992.

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

[27] R. W. Floyd. Assigning meanings to programs. Mathematical aspects of computer science, 19(19-32):1, 1967.

[28] Z. Fu and Z. Su. XSAT: A fast floating-point satisfiability solver. In Proceedings of the 26th International Conference on Computer Aided Verification, CAV’16, Toronto, Ontario, Canada, 2016.

[29] Z. Fu, Z. Bai, and Z. Su. Automated backward error analysis for numerical code. In Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA’15, pages 639–654, Pittsburgh, PA, USA, 2015.

[30] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, pages 213–223, 2005.

[31] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. Monatshefte für mathematik und physik, 38(1):173–198, 1931.

[32] D. Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys (CSUR), 23(1):5–48, 1991.

[33] E. Goubault, S. Le Roux, J. Leconte, L. Liberti, and F. Marinelli. Static analysis by abstract interpretation: A mathematical programming approach. Electronic notes in theoretical computer science, 267(1):73–87, 2010.

[34] S. Heule, M. Sridharan, and S. Chandra. Mimic: Computing models for opaque code. In Proceedings of the 16th Joint Meeting on Foundations of Software Engineering, pages 710–720, 2015.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, Oct. 1969.

[36] J. C. King. Symbolic execution and program testing. Commun. ACM, 19(7):385–394, July 1976.

[37] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. SCIENCE, 220(4598):671–680, 1983.

[38] B. Korel. Automated software test data generation. IEEE Trans. Softw. Eng., 16(8):870–879, Aug. 1990.

[39] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary cover-result criteria for test generation from formal models. In Proceedings of the 15th International Symposium on Software Reliability Engineering,ISSRE ’04, pages 139–150, Washington, DC, USA, 2004.

[40] J. R. Koza. Genetic programming: On the programming of computers by means of natural selection, volume I. MIT press, 1992.

[41] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. Journal of Systems and Software, 83(12):2379–2391, 2010.

[42] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux. FloPSy: Search-based floating point constraint solving for symbolic execution. In Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems, ICTSS’10, pages 142–157, Berlin, Heidelberg, 2010.

[43] K. Lakhotia, M. Harman, and H. Gross. AUSTIN: An open source tool for search based software testing of C programs. Information and Software Technology, 55(1):112–125, 2013.

[44] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO ’04, pages 75–86, Washington, DC, USA, 2004.
[45] D. Leitner, C. Chakravarty, R. Hinde, and D. Wales. Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms. *Phys. Rev. E*, 56:363, 1997.

[46] Z. Li and H. A. Scheraga. Monte Carlo minimization approach to the multiple-minima problem in protein folding. *Proceedings of the National Academy of Sciences of the United States of America*, 84(19): 6611–6615, 1987.

[47] L. Liberti, S. Le Roux, J. Leconte, and F. Marinelli. Mathematical programming based debugging. *Electronic Notes in Discrete Mathematics*, 36:1311–1318, 2010.

[48] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005*, pages 5–20, Edinburgh, UK, 2005.

[49] P. McMinn. Search-based software test data generation: A survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[50] A. M. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *18th IEEE International Conference on Automated Software Engineering, ASE 2003*, pages 164–173, Montreal, Canada, 2003.

[51] B. Meyer. Seven principles of software testing. *IEEE Computer*, 41(8): 99–101, 2008.

[52] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.*, 2(3):223–226, May 1976.

[53] M. Minoux. *Mathematical programming: Theory and algorithms*. Wiley, New York, 1986.

[54] G. J. Myers. The art of software testing. pages I–XV , 1–234, 2004.

[55] M. Minoux. *Mathematical Programming: Theory and Algorithms*. Wiley, New York, 1986.

[56] W. Rudin. *Principles of mathematical analysis*. McGraw-Hill, New York, 3rd edition, 1976.

[57] E. Schlufta, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ‘14*, pages 53–64, New York, NY, USA, 2014.

[58] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005.

[59] J. P. M. Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design, ICCAD ‘96*, pages 220–227, Washington, DC, USA, 1996.

[60] M. Souza, M. Borges, M. d’Amorim, and C. S. Pitáreau. CORAL: Solving complex constraints for symbolic pathfinder. In *NASA Formal Methods Symposium*, pages 359–374, 2011.

[61] M. Tawarmalani and N. V. Sahinidis. Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. *Mathematical programming*, 99(3):563–591, 2004.

A. **Untested Programs in Fdlibm**

The programs from the freely distributed math library Fdlibm 5.3 [3] are used as our benchmarks. Tab. 5 lists all untested programs and functions, and explains the reason why they are not selected. Three types of the functions are excluded from our evaluation. They are (1) functions without any branch, (2) functions involving non-floating-point input parameters, and (3) static C functions.

B. **Implementation Details**

As a proof-of-concept demonstration, we have implemented Algo. 1 in the tool CoverMe. This section presents the implementation and technical details omitted from the main body of the paper.

B.1 **Frontend of CoverMe**

The frontend implements Step 1 and Step 2 of Algo. 1. CoverMe compiles the program under test f00 to LLVM IR with Clang [2]. Then it uses an LLVM pass [6] to inject assignments in f00. The program under test can be in any LLVM-supported language, e.g., Ada, the C/C++ language family, or Julia. Our current implementation accepts C code only.

Fig. 9 illustrates f00 as a function of signature type_t f00 (type_t x1, type_t x2, ...). The return type (output) of the function, type_t, can be any kind of types supported by C, whereas the types of the input parameters, type_t x1, type_t x2, ..., are restricted to double or double*. We have explained the signature of pen in Def. 3.7. Note that CoverMe does not inject pen itself into f00, but instead injects assignments that invoke pen. We implement pen in a separate C++ file.

The frontend also links F00.1 and F00.2 with a simple program loader to generate a shared object file libr.so, which is the outcome of the frontend. It stores the representing function in the form of a shared object file (.so file).

B.2 **Backend of CoverMe**

The backend implements Step 3 of Algo. 1. It invokes the representing function via libr.so. The kernel of the backend is an external MCMC engine. It uses the off-the-shelf implementation known as BasinHopping from the Scipy Optimization package [7]. Basinhopping takes a range of input parameters. Fig. 9 shows the important ones for our implementation basinhopping(f, sp, n_iter, call_back), where f refers to the representing function from libr.so, sp is a starting point as a Python NumPy array, n_iter is the iteration number used in Algo. 1 and call_back is a client-defined procedure. Basinhopping invokes call_back at the end of each iteration (Algo. 1, Lines 24–34). The call_back procedure allows CoverMe to terminate if it saturates all branches. In this way, CoverMe does not need to wait until passing all n_start iterations (Algo. 1, Lines 8–12).
| Program        | Entry function                      | Explanation                  |
|---------------|-------------------------------------|------------------------------|
| e_gamma_r.c   | ieee754_gamma_r(double)             | no branch                    |
| e_gamma.c     | ieee754_gamma(double)               | no branch                    |
| e_j0.c        | pzero(double)                       | static C function            |
| e_j1.c        | qzero(double)                       | static C function            |
| e_jn.c        | pone(double)                        | static C function            |
| e_jn_r.c      | qone(double)                        | static C function            |
| e_lgamma_r.c  | ieee754_lgammar_r(double, int*)     | unsupported input type       |
| e_lgamma.c    | ieee754_lgamma(double)              | no branch                    |
| k_rem_pio2.c  | kernel_rem_pio2(double*, double*, int, const int*) | unsupported input type |
| k_sin.c       | kernel_sin(double, double, int)     | unsupported input type       |
| k_standard.c  | kernel_standard(double, double, int) | unsupported input type |
| k_tan.c       | kernel_tan(double, double, int)     | unsupported input type       |
| s_copysign.c  | copysign(double)                    | no branch                    |
| s_fabs.c      | fabs(double)                        | no branch                    |
| s_finite.c    | finite(double)                      | no branch                    |
| s_flexp.c     | frexp(double, int*)                 | unsupported input type       |
| s_lxexp.c     | ldexp(double, int)                  | unsupported input type       |
| s_lib_version.c | lib_version(double)               | no branch                    |
| s_matherr.c   | matherr(struct exception*)         | unsupported input type       |
| s_scalbn.c    | scalbn(double, int)                 | unsupported input type       |
| s_signgam.c   | signgam(double)                     | no branch                    |
| s_significand.c | significand(double)               | no branch                    |
| w_acos.c      | acos(double)                        | no branch                    |
| w_acosh.c     | acosh(double)                       | no branch                    |
| w_asin.c      | asin(double)                        | no branch                    |
| w_atan2.c     | atan2(double, double)               | no branch                    |
| w_atanh.c     | atanh(double)                       | no branch                    |
| w_cosh.c      | cosh(double)                        | no branch                    |
| w_exp.c       | exp(double)                         | no branch                    |
| w_fmod.c      | fmod(double, double)                | no branch                    |
| w_gamma_r.c   | gamma_r(double, int*)               | no branch                    |
| w_gamma.c     | gamma(double, int*)                 | no branch                    |
| w_hypot.c     | hypot(double, double)               | no branch                    |
| w_j0.c        | j0(double)                          | no branch                    |
| w_j1.c        | j1(double)                          | no branch                    |
| w_jn.c        | jn(double)                          | no branch                    |
| w_lgamma_r.c  | lgamma_r(double, int*)              | no branch                    |
| w_lgamma.c    | lgamma(double)                      | no branch                    |
| w_log.c       | log(double)                         | no branch                    |
| w_log10.c     | log10(double)                       | no branch                    |
| w_pow.c       | pow(double, double)                 | no branch                    |
| w_remainder.c | remainder(double, double)           | no branch                    |
| w_scalb.c     | scalb(double, double)               | no branch                    |
| w_sinh.c      | sinh(double)                        | no branch                    |
| w_sqrt.c      | sqrt(double)                        | no branch                    |
Dealing with Comparison between Non-floating-point Expressions (Relaxing Def. 3.1(b)). We have encountered situations where a conditional statement invokes a comparison between non-floating-point numbers. CoverMe handles these situations by first promoting the non-floating-point numbers to floating-point numbers and then injecting\(^{1}\)pen as described in Algo. 1. For example, before a conditional statement like\(^{1}\)if (xi op yi) where xi and yi are integers, CoverMe injects\(^{1}\)r = pen (i, op, (double) x, (double) y);. Note that such an approach does not allow us to handle data types that are incompatible with floating-point types, e.g., conditions like\(^{1}\)if (p != Null), which CoverMe has to ignore.

Dealing with Infeasible Branches (Relaxing Def. 3.1(e)). Infeasible branches are branches that cannot be covered by any input. Attempts to cover infeasible branches are useless and time-consuming. Detecting infeasible branches is a difficult problem in general. CoverMe uses a simple heuristic to detect and ignore infeasible branches. When CoverMe finds a minimum that is not zero, that is,\(^{1}\)F00.R(x) > 0, CoverMe deems the unvisited branch of the last conditional to be infeasible and adds it to Saturate, the set of unvisited and deemed-to-be infeasible branches.

Imagine that we modify\(^1\)l1 of the program F00 in Fig. 3 to the conditional statement\(^1\)if (y == -1). Then the branch\(^1\)l1 becomes infeasible. We rewrite this modified program below and illustrate how we deal with infeasible branches.

\[\begin{align*}
  l0: & \text{ if } (x <= 1) \{x++\}; \\
  & y = \text{square}(x); \\
  l1: & \text{ if } (y == -1) \{\ldots\}
\end{align*}\]

where we omit the concrete implementation of square.

Let F00.R denote the representing function constructed for the program. In the minimization process, whenever CoverMe obtains\(^1\)x such that\(^1\)F00.R(x) > 0, CoverMe selects a branch that it regards infeasible. CoverMe selects the branch as follows: Suppose\(^1\)x* exercises a path\(^1\)τ whose last conditional statement is denoted by l*, and, without loss of generality, suppose\(^1\)y* is passed through by\(^1\)τ, then CoverMe regards\(^1\)y* as an infeasible branch.

In the modified program above, if\(^1\)l1 has been saturated, the representing function evaluates to\(^1\)((y + 1)^2 + (y + 1)^2 + 1$, where\(^1\)y equals to the non-negative square(x). Thus, the minimum point\(^1\)x* must satisfy\(^1\)F00.R(x*) > 0 and its triggered path ends with branch\(^1\)l1. CoverMe then regards\(^1\)l1 as an infeasible branch.

CoverMe then regards the infeasible branches as already saturated. It means, in line 12 of Algo. 1, CoverMe updates Saturate with saturated branches and infeasible branches (more precisely, branches that CoverMe regards infeasible).

The presented heuristic works well in practice (See Sect. 5), but we do not claim that our heuristic always correctly detects infeasible branches.

Dealing with Function Calls. By default, CoverMe injects\(^1\)r = pen, only in the entry function to test. If the entry function invokes other external functions, they will not be transformed. For example, in the program F00 of Fig. 3, we do not transform square(x). In this way, CoverMe only attempts to saturate all branches for a single function at a time.

However, CoverMe can also easily handle functions invoked by its entry function. As a simple example, consider:

\[
\begin{align*}
  \text{void F00(double x)} & \{ \text{G00(x);} \} \\
  \text{void G00(double x)} & \{ \text{if } (\sin(x) == 0.99) \ldots \}
\end{align*}
\]

If CoverMe aims to saturate F00 and G00 but not sin, and it sets F00 as the entry function, then it instruments both F00 and G00. Only G00 has a conditional statement, and CoverMe injects an assignment on\(^1\)r in G00.

B.3 Technical Details

Sect. 3 assumes Def. 3.1(a-c) for the sake of simplification. This section discusses how CoverMe relaxes the assumptions when handling real-world floating-point code. We also show how CoverMe handles function calls at the end of this section.

Dealing with Pointers (Relaxing Def. 3.1(a)) We consider only pointers to floating-point numbers. They may occur (1) in an input parameter, (2) in a conditional statement, or (3) in the code body but not in the conditional statement.

CoverMe inherently handles case (3) because it is execution-based and does not need to analyze pointers and their effects. CoverMe currently does not handle case (2) and simply ignores these conditional statements by not injecting pen before them.

Below we explain how CoverMe deals with case (1). A branch coverage testing problem for a program whose inputs are pointers to doubles, can be regarded as the same problem with a simplified program under test. For instance, finding test inputs to cover branches of program void F00 (double x) \{ if (x <= 1) \ldots \} can be reduced to testing the program void F00_with_no_pointer (double x) \{ if (x <= 1) \ldots \}. CoverMe transforms program F00 to F00_with_no_pointer if a F00’s input parameter is a floating-point pointer.

Figure 9: CoverMe Implementation.