Interval Constraint-Based Mutation Testing of Numerical Specifications

Clothilde Jeangoudoux
jeangoudoux@mpi-sws.org
MPI-SWS
Germany

Eva Darulova
eva@mpi-sws.org
MPI-SWS
Germany

Christoph Lauter
clauter@alaska.edu
University of Alaska Anchorage
USA

ABSTRACT

Mutation testing is an established approach for checking whether code satisfies a code-independent functional specification, and for evaluating whether a test set is adequate. Current mutation testing approaches, however, do not account for accuracy requirements that appear with numerical specifications implemented in floating-point arithmetic code, but which are a frequent part of safety-critical software. We present Magneto, an instantiation of mutation testing that fully automatically generates a test set from a real-valued specification. The generated tests check numerical code for accuracy, robustness and functional behavior bugs. Our technique is based on formulating test case and oracle generation as a constraint satisfaction problem over interval domains, which soundly bounds errors, but is nonetheless efficient. We evaluate Magneto on a standard floating-point benchmark set and find that it outperforms a random testing baseline for producing useful adequate test sets.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Mathematics of computing → Interval arithmetic**; • **Theory of computation → Constraint and logic programming**.

KEYWORDS

mutation testing, constraint programming, functional specification, floating-point arithmetic, interval arithmetic

ACM Reference Format:
Clothilde Jeangoudoux, Eva Darulova, and Christoph Lauter. 2021. Interval Constraint-Based Mutation Testing of Numerical Specifications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460319.3464808

1 INTRODUCTION

Mutation testing [37] is an established white-box approach for checking whether code correctly implements a specification. Test cases are generated from mutations of the specification that simulate common programming mistakes that should be guarded against. The aim is to develop a test suite that will detect all such faults (or *kill* all mutants), i.e. include at least one test to show that the target code's behavior differs from the mutant's.

Mutation testing has been successful in a number of domains, such as real-time embedded systems [25], hardware verification in the micro-electronics industry [60] or more recently performance testing [23]. Unfortunately, these techniques are not suitable for numerical specifications that appear frequently for instance in automotive or avionics applications, because they ignore rounding errors due to finite-precision arithmetic.

Such numerical functional specifications are given in terms of real values, but their implementations are (of necessity) written in terms of finite precision such as floating-point arithmetic [2]. Effectively, this means that the code can only implement the functional specification *approximately*: every arithmetic operation inherently suffers from rounding errors that accumulate throughout the computation. To rigorously test finite-precision code, these rounding errors have to be taken into account when generating test oracles and test inputs for common (syntactic) errors, such as incorrect arithmetic operators. In addition, a meaningful test set should also check whether the code is implemented with sufficient precision, i.e. whether it computes accurate enough results.

We present the first mutation testing technique for numerical specifications with accuracy requirements. Given a real-valued specification as an arithmetic expression together with an accuracy requirement, our algorithm generates a test set which adequately checks for common programming mistakes, such as wrong constants or arithmetic operators, but which also checks whether the implementation is sufficiently accurate. The accuracy requirement is given as a relative error, for instance specifying that Equation 1 should be evaluated with relative accuracy of 10^{-7} with respect to the exact result computed with infinite precision. Our approach is agnostic to the actual finite precision used in the code to be tested, and can thus be used to test floating-point or fixed-point arithmetic [61] implementations.

Automated test oracle and test case generation for numerical specifications is nontrivial, because automated reasoning, e.g. in the form of decision procedures, is expensive for nonlinear real [56] and floating-point arithmetic [50], and exact reasoning about elementary functions, such as sine and exponential, is undecidable [45].

Our key technical contribution is to introduce constraint programming [1] *over interval domains* [41] as a foundation for mutation testing of numerical specifications. Interval arithmetic allows our algorithm to handle specifications with elementary functions and bounded input domains, and to soundly capture accuracy errors, while at the same time the resulting nonlinear constraints are efficiently solved by existing tools [33].

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07.
https://doi.org/10.1145/3460319.3464808

Given a target specification, our algorithm first generates a number of mutants from three categories of faults: functional behavior, robustness, and accuracy. These, respectively, test whether the code implements the correct arithmetic expression, whether it supports the correct input range, and whether it is implemented with sufficient precision. While the first category of mutants is standard, the latter two are specific to the domain of finite-precision implementations.

Given a test input, we show how to compute a *test oracle* that checks whether a particular implementation passes the test, correctly accounting for uncertainties due to finite-precision rounding errors. Without access to the actual implementation (as in this setting) it is fundamentally not feasible to check whether a test input is *guaranteed* to discover when the implementation is not sufficiently accurate. It is, however, possible to identify test inputs that are *likely* to find accuracy issues, which is what our approach does. Our interval-based test oracle computation can compute oracles for test inputs obtained with any test generation procedure, including random testing.

We furthermore show how to encode *test generation* for numerical mutation testing as an interval constraint problem, and how to solve it efficiently. Such a principled test generation procedure can find low-probability test inputs that are hard to find with random testing. Furthermore, our procedure is deterministic and thus applicable in the strict certification process of safety-critical software [52] that disallows random testing.

We implement our algorithm in a prototype tool called Magneto in the Julia programming language [10], using RealPaver [33] as the back-end constraint solver. We evaluate Magneto on 69 benchmarks from the standard floating-point benchmark suite FPBench [19]. Magneto achieves an average mutation score of 0.88 with an average running time of 77 seconds. Compared to random testing, Magneto achieves a higher mutation score on 65% of the benchmarks. In particular, Magneto outperforms random testing by 28% when checking for accuracy errors and 14% for input domain errors.

Contributions. In summary, this paper makes the following contributions:

- a test oracle computation for numerical specifications (Section 4),
- a novel, fully automated, algorithm for mutation testing-based test generation of numerical specifications with accuracy requirements (Section 5),
- an implementation in the tool Magneto (Section 6) that is available as open-source [1], and
- an evaluation showing that Magneto generates adequate test sets which are better than a random testing baseline (Section 7).

2 OVERVIEW

We first give a high-level overview of the challenges and Magneto's solution using an example. The formula

$$r = 5 \cdot \frac{x_1 - \frac{x_3}{x_4}}{x_2} \tag{1}$$

defines the extracted air flow of a helicopter engine, and will serve as the specification given to Magneto that we will denote by $\mathcal{S}_r$. Such an expression would, for instance, appear as part of a unit test

[1] https://github.com/clothildejeangoudoux/Magneto

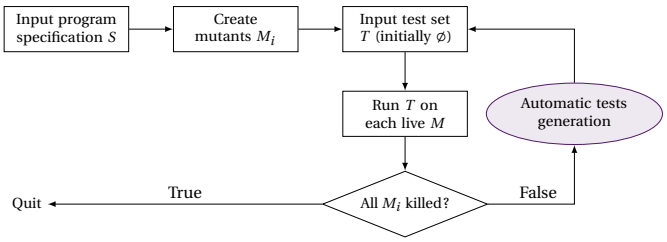


Figure 1: Mutation Testing for Automated Test Generation

in a safety-critical application. This specification is real valued, i.e. $x_i, r \in \mathbb{R}$, and we further assume that a relative accuracy bound of $\varepsilon = 10^{-7}$ for the finite-precision implementation is given. Finally, we will assume the following input domains for the variables:

$$x_1, x_3 \in [10^{-7}, 10], \quad x_2, x_4 \in [10^{-7}, 1] \tag{2}$$

over which the formula is meaningful. (Our approach can also work with unbounded domains.)

The goal of mutation testing [37] and of Magneto is to generate an adequate test set that will check an implementation of $\mathcal{S}_r$ against common programming mistakes (we provide more background in Section 3.1). Figure 1 shows Magneto's high-level work flow, which starts by generating a set of mutants. To obtain a mutant $\mathcal{M}$, Magneto injects a fault, e.g. a syntactic change that corresponds to a possible programming error, into the specification $\mathcal{S}$. Magneto then iteratively generates test inputs that distinguish between $\mathcal{S}$ and the mutants. We say that the verdict of a test is `KO`, if a test input found an injected bug in a mutant (*killed* the mutant), and `OK` otherwise. The goal is thus to kill all the mutants, i.e. generate tests which will result in `KO` for at least one mutant.

Let us ignore the accuracy requirement initially and assume that both the specification and implementation are real-valued. One of the mutations that Magneto applies is changing a binary arithmetic operator for another, simulating a possible typo in the implementation. For example, changing the subtraction into an addition in Equation 1 results in a mutant $\mathcal{M}_r$:

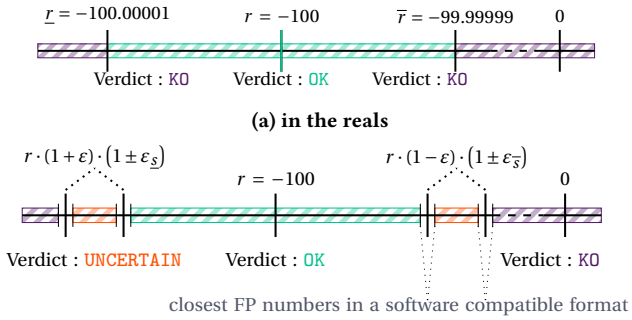
$$r_M = 5 \cdot \frac{x_1 + \frac{x_3}{x_4}}{x_2} \tag{3}$$

Initially, the test set T is empty, so that the mutant $\mathcal{M}_r$ lives. The specification's input domain (Equation 2) defines a set of valid values for r such that $r \in [-5 \cdot 10^{15}, -5 \cdot 10^{8}]$ as illustrated in green in Figure 2a. Similarly, the result of the mutant r_M over those input intervals can take a value in some (other) interval, here, $r_M \in [5.49999 \cdot 10^{-7}, 5.00001 \cdot 10^{14}]$. In order to distinguish between $\mathcal{S}_r$ and $\mathcal{M}_r$, Magneto thus has to identify an input, which is in the result interval of $\mathcal{M}_r$, but not in $\mathcal{S}_r$, shown in purple with verdict `KO` in Figure 2a.

Magneto performs test generation by encoding the constraint that a distinguishing test output r is in the valid range of the specification, but not in the range of the mutant:

$$r \in 5 \cdot \frac{x_1 - \frac{x_3}{x_4}}{x_2} \wedge r \notin 5 \cdot \frac{x_1 + \frac{x_3}{x_4}}{x_2}$$

This constraint is interval-valued, i.e. the input variables x_i are bounded by intervals. With some abuse of notation, we write $r \in$



(a) in the reals

(b) in finite precision

Figure 2: Different verdict scenarios

$f(x_1, x_2, x_3, x_4)$ to mean that the value of r is constrained to be in the image of f (for all possible input valuations).

Magneto uses the Realpaver [33] off-the-shelf solver to generate a solution to this constraint problem:

$$x_1 = 10^{-2}, x_2 = 10^{-1}, x_3 = 10^{-4}, x_4 = 10^{-1},$$

which when executed on $\mathcal{S}_r$ results in $r = -50000.5$, but when executed on $\mathcal{M}_r$, gives $r_M = 50000.5$.

When both the specification and the implementation are real-valued, this is sufficient. However, in practice, we need to also consider the finite precision of the implementation, which leads to rounding errors in computing r_M, which in turn are bounded by the accuracy specification ($\varepsilon = 10^{-7}$). An additional complicating factor is that the constraint solver itself is using floating-point arithmetic. This is unavoidable, as our specifications may include transcendental functions such as sine and exponential which cannot be efficiently represented exactly. In summary this means that checking whether a test case reliably kills a mutant is nontrivial in the presence of floating-point arithmetic as we *cannot compute the outputs for the* `OK` *and* `KO` *verdicts exactly*. Note that this is the case regardless of how the tests were generated.

Our solution is to use interval arithmetic to not compute just a single valid output, but rather valid and invalid *domains* that soundly over-approximate the rounding errors due to finite precision. This over-approximation leads to a 'gap' between the valid and invalid domains, where we cannot decide whether an output is valid or not; we denote this area as `UNCERTAIN` in Figure 2b. Magneto takes that uncertain area into consideration such that test generation does not return test inputs giving an output in this area.

We need to consider inaccuracies due to finite precision in generating test verdicts, but we also need to generate tests that specifically check whether the implementation is computed with sufficient accuracy, even in the absence of other faults. Magneto checks for sufficient accuracy by introducing an accuracy mutation, for instance by setting the accuracy bound on the mutant to a higher value $\varepsilon_M = 10^{-6}$ (allowing for a larger error). However, since we do not have any syntactic information about the code, it is not possible to find inputs which can be returned by $\mathcal{M}$, but not by $\mathcal{S}$ since the valid range of $\mathcal{S}$ is, by necessity, included in the valid range of $\mathcal{M}$. We can, however, test whether those ranges overlap more than a certain amount. By doing so, we can generate tests for which the probability of capturing code inaccuracies are high.

3 BACKGROUND

3.1 Mutation Testing

Mutation testing [37] is an approach for generating a test set and evaluating its ability to detect faults in an implementation. It is based on the idea of deliberately injecting the faults one wants to check for, and then check that the test set can correctly identify them. This approach can be used to evaluate an existing test set or, as in our setting, generate a test set from a specification to check an independently developed implementation. That is, the test set is developed without access to the code to be tested, which is important for instance for certification purposes in avionics [52]. Since injection of faults requires knowledge of the full specification, mutation testing is a white-box testing technique.

Injecting a fault into the original program, or mutating it, results in a mutant. The test set is then executed over the set of mutants to verify its quality. If a test successfully finds an injected fault, we say the test verdict is `KO` and the test killed the mutant. If a mutant passes the test, i.e. has the correct behavior according to the specification, the test verdict is `OK`.

In theory, we say that a test set is *adequate* if all incorrect versions of the program were successfully detected by the test set. In practice, generating adequate test sets or detecting that a given test set is adequate is an undecidable problem [11].

Mutation testing focuses on faults based on the Competent Programmer Hypothesis (CPH), which assumes that programmers tend to develop programs that are close to their correct version. Therefore, faults in such programs can be fixed by small syntactic changes. This assumption makes it possible to aim to find a *relatively adequate* test set, that will successfully find a finite number of incorrect versions of the program. In practice, the measure of relative adequacy of the program to a specific test set is called the mutation adequacy score, or *mutation score* [24]. The mutation score is the ratio of the number of mutants that were killed by the test set over the total number of generated mutants.

3.2 Accuracy and Precision

The specifications from which our approach generates test inputs are real-valued, but the implementations that these tests are supposed to verify are inevitably implemented in finite precision and thus necessitate accuracy requirements.

There is a difference between the *precision* of an implementation, which is defined by the number of bits used by the finite-precision arithmetic, and the *accuracy* of a computation, which in this context measures how far away a computed value is from the ideal, real-valued output. An implementation may compute an output with 64 bits of precision, but if many of those digits are wrong it will nonetheless have low accuracy.

We measure accuracy as a relative error between an ideal value x and the computed value $\tilde{x}$:

$$\varepsilon = \frac{\tilde{x} - x}{x}, \text{ if } x \neq 0. \tag{4}$$

In this paper, we focus on accuracy, because the actual implementation and thus its precision is not available *by design*. That is, the goal of our generated tests is to check, among other faults, whether the implementation is computing its results sufficiently accurately, without considering the actual precision of the implementation.

We provide basic background on floating-point arithmetic, which is one of the most common representations [29, 45] for finite precision, and in which our own approach is implemented. Note that the implementations that Magneto generates tests for can be implemented in a different finite precision representation, e.g. in fixed-point arithmetic.

The IEEE 754 standard [2] defines floating-point arithmetic in two radices: 2 and 10. We focus on binary floating-point arithmetic, as hardware support for decimal is limited. In the standard, a binary floating-point number x is described by

$$x \quad = \quad (-1)^s \cdot m \cdot 2^e,$$

where $s \in \{0, 1\}$ is the sign bit, $e \in [e_{\min}, e_{\max}]$ is the exponent, and m is the significand of precision p such that $0 \leq |m| < 2$. Commonly used precisions are binary32 (single precision), with $p = 8$ and $e_{\max} = -e_{\min} + 1 = 2^{8-1} - 1$ and binary64 (double precision), $p = 53$ and $e_{\max} = -e_{\min} + 1 = 2^{11-1} - 1$.

Additionally, the special values Infinity and Not-a-Number (NaN) signal overflow, and mathematically ill-defined operations such as square root of a negative number, respectively. In this paper, we consider specifications that do not lead to special values, i.e. consider them as errors.

Finite precision means that not all real numbers are representable in floating-point arithmetic and thus need to be rounded. Under the default rounding mode of rounding to nearest, the IEEE 754 standard defines that the result of every arithmetic operation needs to be equal to the one obtained if the operation were performed in infinite precision and then rounded. Transcendental functions such as sine and exponential are implemented in libraries, which specify the corresponding, often larger, rounding error bound. While the errors of individual arithmetic operations are often small and are bounded by the IEEE 754 standard, they propagate through a program in often unintuitive ways and can accumulate.

3.3 Interval Arithmetic

Interval arithmetic [42] allows to represent and compute with sets of values, such as the uncertain results of finite-precision arithmetic. A closed interval represents a range of values by its lower and upper bound: $\mathbf{X} = [\underline{x}, \overline{x}] := \{x \mid \underline{x} \leq x \leq \overline{x}\}$. Binary arithmetic operations over intervals $\mathbf{X}$ and $\mathbf{Y}$ soundly enclose all possible results of the operation assuming that the input arguments are in $\mathbf{X}$ and $\mathbf{Y}$:

$$\mathbf{X} \circ \mathbf{Y} \supseteq \left\{ x \circ y \mid x \in [\underline{x}, \overline{x}] = \mathbf{X}, y \in [\underline{y}, \overline{y}] = \mathbf{Y} \right\} \tag{5}$$

where $\circ \in \{+, -, \times, \div\}$. Unary operations are analogous.

Interval arithmetic can in particular be used to soundly bound real values that are not exactly representable in floating-point arithmetic, such as 0.1 or π. For this, we can use intervals with floating-point bounds, where $\underline{x}$ is the closest representable smaller and $\overline{x}$ is the closest representable larger floating-point number, i.e. $\underline{x} \leq 0.1 \leq \overline{x}$.

4 RIGOROUS TEST ORACLES

In this section, we introduce constraint programming over interval domains as a foundation for rigorous input specification and test oracle computation for mutation testing of numerical specifications.

We will use the following toy specification $\mathcal{S}$ as running example to illustrate our explanations:

$$(\mathcal{S}) \qquad y \in x^3 \cdot (1 + \varepsilon), \quad |\varepsilon| \leq 10^{-4}, \quad x \in [-1.5, 1.5] \tag{6}$$

4.1 Constraints as Input Specification

The input to our mutation testing algorithm is a formal specification $\mathcal{S}$. We choose constraint programming [1] over interval domains as a formalism to represent requirements over real numbers. Constraint programming is a paradigm for expressing relationships between different entities in the form of constraints that must be satisfied. Constraint programming is essentially characterized by two phases: *modeling* involves the identification and formalization of a problem, and *solving*, which searches for a solution of the constraints. Constraints over intervals allow us to capture, on one hand, specifications that are valid only over a bounded input domain, and on the other hand, uncertainties due to finite-precision arithmetic.

We follow the notation of [6] and model a Constraint Satisfaction Problem (CSP) as a triplet $(\mathcal{X}, \mathcal{D}, C)$, such that

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of variables,
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of definition domains, such that a domain D_i defines all the possible values for a variable x_i, with $i = 1, \ldots, n$,
- $C = \{c_1, ..., c_m\}$ is a finite set of constraints, restricting the values that can be assigned simultaneously to the variables.

For an input specification $\mathcal{S}$, the input and output variables of the specification define the variables x_i of the CSP. The definition domains of the input variables are given by the range specified in the requirement. For output variables, or if no specific domain is specified for a variable, they are represented in the CSP over $[-\infty, \infty]$. The functional requirement of the specification (i.e. the equation to be computed) determines the constraints. In order to take into account the accuracy requirement, we multiply the functional requirement equation by $(1 + \varepsilon)$ and restrict ε to have a magnitude smaller than the given maximum relative error.

Example. Our running example specification in Equation 6 is expressed formally as the following CSP:

$$\begin{aligned} \text{Variables :} \quad & x \in [-1.5, 1.5], \quad y \in [-\infty, \infty], \\ & \varepsilon \in [-10^{-4}, 10^{-4}], \\ \text{Constraint :} \quad & c : y \in x^3 \cdot (1 + \varepsilon) \end{aligned}$$

We will use the less verbose format of Equation 6 in the rest of paper, whenever it is suitable.

4.2 Mutant Generation

Next, we define the mutant generation in Magneto, as test oracles depend on the mutants. Magneto automatically and randomly generates a set of mutants using the set of mutation rules given in Table 1. These mutations are relevant to test generation for numerical software because they emulate (a subset of) simple errors a software engineer may introduce. To generate a mutant, Magneto

Table 1: Set of mutations considered in Magneto

Operator	Description
`accuracy`	decrease the accuracy by increasing the error bound ε
`bounds`	replace the bounds of an input variable with new values
`constant`	replace the value of a constant with a new value
`variable`	swap two variables
`unary`	replacement of an unary operator with another $\left(\sqrt{}, \sin, \cos, \tan, \exp, \log\right)$
`binary`	replacement of an binary operator with another $(+, -, \times, \div, \max, \min)$
`add`	add an operator
`del`	delete an operator

selects a mutation from the list and applies it to one randomly selected subexpression or parameter of the specification.

We only generate first order mutants, meaning a single fault is injected in each mutant. However, higher order mutations [36], which inject two or more faults could be added to Magneto in order to further reduce test effort.

In the context of certified software, one of the requirements of software verification is traceability [17] between the program specification, test objectives and the tests. For numerical software, test objectives form three categories [52], which we use as the basis for the selection of the mutation operators in Table 1:

- precision: verifies that the implementation uses sufficient precision, as defined in the accuracy requirement of the specification (mutation `accuracy`),
- robustness: verification that the software can proceed safely under abnormal circumstances, such as input values outside of their bounds and abnormal behavior in the output (mutations `bounds`, `add`, `del`),
- functional behavior: verifies the absence of errors in the functional behavior (mutations `constant`, `variable`, `unary`, `binary`).

Most of these mutation operators aim at capturing common syntactic mistakes. However, using interval constraints as specification allowed us to introduce the `accuracy` and `bounds` mutation operators, which are specific to the domain of finite-precision computations.

Example. Applying the mutation `accuracy` to the specification $\mathcal{S}$ from our running example (Equation 6), decreases the accuracy by increasing the corresponding ε bound. It thus increases the error bound ε such that $\varepsilon < \varepsilon_M < 1$ to create the mutant:

$$(\mathcal{M}_1) \qquad y \in x^3 \cdot (1+\varepsilon_m),\ |\varepsilon_m| \leq 10^{-3},\ x \in [-1.5, 1.5] \tag{7}$$

An example of the robustness test objective is to use the mutation `bounds` to produce the mutant

$$(\mathcal{M}_2) \qquad y \in x_m^3 \cdot (1+\varepsilon),\ |\varepsilon| \leq 10^{-4},\ x_m \in [-1.5, 1.6]$$

The mutation modifies the bounds of the variable x to form the variable x_m. This mutation operator relates to a certain type of real bug, where the specified bounds of a numerical requirement inside a bigger specification are now met because some numerical error occurred before in the code. Hence it is important to know if the numerical requirement under test is sensitive to changes of those bounds.

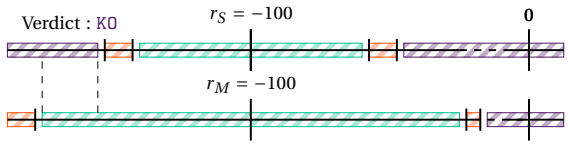


Figure 3: Accuracy mutant comparison with test oracle

4.3 Computing Test Oracles

Given a test input, either obtained through random testing or with a principled approach as in Section 5, we need to compute a corresponding test oracle that determines whether an implementation passes the test, resp. whether a mutant is killed. The test oracles for a given test set thus allow us to evaluate the adequacy of a test set.

Since we do not have an implementation and its precision available (by design), we cannot simply evaluate the specification and the mutant expressions on the test input to obtain single values as a test oracle—we only have the overall uncertainty expressed as a relative error on the *real-valued* result. Computing the ideal real-valued result, however, is not practically feasible in general because of transcendental functions that do not have a finite representation.

We compute a rigorous test oracle by evaluating the test inputs over the constraint specification and the constraint mutants with interval arithmetic, producing two types of domains: valid and invalid. The verdict of a test input evaluated over a mutant is `KO` if the mutant's valid range is included in the invalid range of the specification and the mutant is killed. Similarly, the verdict is `OK` if the mutant's valid range is included in the valid range of the specification; in this case the mutant remains alive.

We use interval arithmetic with lower and upper bounds implemented with MPFR, i.e. floating-point arithmetic with a high working precision. Due to rounding errors in the computation of the test oracle domains, and the over-approximation of interval arithmetic, there is a 'gap' between the valid and invalid ranges. If the mutant's valid range overlaps with that gap, or if the mutant invalid range overlaps with the valid range of the specification, the verdict is `UNCERTAIN`. We say that our test oracle is rigorous because the `UNCERTAIN` verdict does not kill the mutants.

Example. For the requirement $\mathcal{S}$ (Equation 6), two values y_1 and y_2 are computed from the input input x_i with interval arithmetic, such that $y_1 = x_i^3 \cdot (1+\varepsilon) = [\underline{y_1}, \overline{y_1}]$ and $y_2 = x_i^3 \cdot (1-\varepsilon) = [\underline{y_2}, \overline{y_2}]$. The bounds of y_1 and y_2 then become the bounds of the valid and invalid output ranges. That means that, if $y_1 < y_2$, then the valid range will be $[\overline{y_1}, \underline{y_2}]$, and the invalid range $]-\infty, \underline{y_1}[\cup]\overline{y_2}, +\infty[$.

Accuracy Mutation. For the accuracy mutation, the verdict of the test according to our test oracle is always `UNCERTAIN`, since, as shown in Figure 3, the valid range of the specification (top line) is always included in the valid range of the mutant (bottom line). Effectively, generating tests that are guaranteed to capture accuracy errors requires knowledge of the actual code. Instead, we aim at identifying test inputs for which the overlap between the mutant's

valid and the specification's invalid range is 'big'. We consider that overlap big enough when the difference between the values is larger than three order of magnitude of the relative error ε, i.e. if $\varepsilon = 10^{-7}$, the difference must be larger than 10^{-4}.

The width of the specification's valid range essentially measures numerical instability, since it is computed in interval arithmetic with high-precision floating-point bounds. The difference between the mutant's and the specification's valid range is due to the mutant's larger error, which effectively accentuates the instability. The larger the instability of the arithmetic expression on the particular test input is, the larger the (absolute) overlap of the mutant's valid and the specification's invalid domain will be. Further, for accuracy mutations where the difference between ε and ε_M is smaller, the underlying instability has to be larger to pass the test oracle.

Note that when the result of an actual implementation on a test input lies outside of the valid range of the specification, we have *definitely* discovered a bug. Conversely, however, when the result is inside the specification's valid range, we cannot exclude an accuracy bug.

Going back to the example in Equation 1, suppose Magneto generates the test input $x_1 = 49 \cdot 10^{-5}, x_2 = 5 \cdot 10^{-7}, x_3 = 10^{-4}, x_4 = 2 \cdot 10^{-1}$. For those inputs Magneto defines the valid output in the interval $r_S \in]-100.0000099, -99.9999900[$ and the invalid output in $r_s \in]-\infty, 100.0000799[\cup]-99.9999100, +\infty[$, whereas the valid range of the mutant is $r_M \in]-100.00311, -99.998398[$. Figure 3 shows these ranges at the top for the specification and below for the mutant. Since the interval $]-100.00311, -100.0000799[$ represent the overlapping range, and $|-100.00311 + 100.0000799| > 10^{-4}$, the mutant is considered to be killed.

5 PRINCIPLED TEST GENERATION

A naive approach to generate an adequate test set is to use random testing. Random testing over a constraint specification and a set of constraint mutants amounts to evaluating a randomly generated test input with our rigorous test oracle. If the new test kills a live mutant, then it is added to the test set, otherwise the test is discarded and a new test is randomly generated. The procedure stops when all mutants are killed or when a timeout is reached.

While random testing is a straight-forward approach for test generation, it is not allowed for certifying safety-critical software [52], and may fail to kill mutants which require very particular inputs.

This section covers Magneto's algorithm based on constraint programming over interval domains for test generation, first at a high-level and then in more detail.

5.1 High-Level Algorithm

Recall the overall mutation testing algorithm from Figure 1. Magneto first creates a list of mutants (Section 4.2) from the specification $\mathcal{S}$, and then iteratively generates test inputs until all mutants have been killed (or until a time-out) using Algorithm 1. For each test input, Magneto computes a test oracle using the procedure outlined in Section 4.3.

Algorithm 1 for test generation takes as input the specification $\mathcal{S}$ and one mutant $\mathcal{M}$, both represented as a constraint satisfaction problem (CSP) over continuous domains (Section 4.1). A suitable test to kill this mutant would be a test that discriminates the mutant from the specification. The first step of the algorithm is thus to create a new CSP $\mathcal{R}$ (`gen_csp_test`) that defines the set of values that are solutions of $\mathcal{S}$ but not solutions of $\mathcal{M}$ (Section 5.2).

Algorithm 1: Test generation procedure

Input: Program Specification – $\mathcal{S} = \{\mathcal{X}, \mathcal{D}, C\}$
Live Mutant – $\mathcal{M} = \{\mathcal{X}_M, \mathcal{D}_M, C_M\}$
Output: New test case – t

```
1  R ← gen_csp_test (S, M)
2  Δ ← ε_S
3  while Δ > Δ_min do
4  |  r ← solve_csp (R, Δ)
5  |  if No solution is found then
6  |  |  Δ ← Δ / 10.0
7  |  else
8  |  |  t ← pick_test (S, r)
9  |  |  Return t
10 |  end
11 end
12 Return ⊥
```

The procedure `solve_csp` calls a CSP solver in order to solve $\mathcal{R}$, i.e. to find inputs that are solutions of the constraints. The solution itself is also given as a set of intervals. The CSP solver searches for a solution by iteratively and on-demand subdividing the input ranges until it can show that one of the (small) intervals is a solution (Section 5.3) or until it reaches intervals of a minimum width Δ, given as a parameter to the solver. If no solution was found, Δ is decreased, and the solving starts again, but will now take potentially longer as with a smaller Δ, the solver has to perform more subdivisions. Magneto repeats this loop until $\Delta = \Delta_{\min}$, for a predetermined value of minimum interval width, which we discuss further in Section 5.3.

If a solution is found that solves $\mathcal{R}$, then `pick_test` proceeds to select a test input from the produced intervals. It then computes valid and invalid ranges for the result of the specification expression, which determine the test verdict (Section 5.3).

If Algorithm 1 reaches Δ_{min} without finding a solution (line 13), the mutant $\mathcal{M}$ is equivalent or too similar to the specification $\mathcal{S}$ such that the CSP solver cannot distinguish them. We discuss limitations of Algorithm 1 in Section 5.4.

Example. If we apply the `constant` mutation that changes the constant (power) 3 to 2 in specification $\mathcal{S}$ (Equation 6), we obtain the following mutant:

$$(\mathcal{M}_1) \qquad y \in x^2 \cdot (1+\varepsilon), \quad |\varepsilon| \leq 10^{-4}, \quad x \in [-1.5, 1.5] \tag{8}$$

The output of the `gen_csp_test` procedure with this mutant is the constraint

$$\Big(y \in x^3 \cdot (1+\varepsilon)\Big) \wedge \Big(y \notin x^2 \cdot (1+\varepsilon)\Big), \quad |\varepsilon| \leq 10^{-4},\ x \in [-1.5, 1.5].$$

With $\Delta = \varepsilon_{\mathcal{S}} = 10^{-4}$, the solver returns the solution $x \in [0.88008, 0.88012]$ (the values are rounded for presentation purposes). `pick_test` heuristically selects the input $x = 0.8801$, and determines the bounds of the valid range of y to be $\underline{y} = [0.681645, 0.68165]$ and $\overline{y} = [0.68179, 0.68180]$. Thus, the test input

and oracle generated for the mutant in Equation 8 is:

$$x = 0.8801,$$
$$y \in [0.68165, 0.68179] \rightarrow \texttt{OK},$$
$$y \in]-\infty, 0.681645[\cup]0.68180, +\infty[\rightarrow \texttt{KO}.$$

In order to check whether the mutant $\mathcal{M}_1$ is killed, we evaluate it with the input $x = 0.8801$ using interval arithmetic, resulting in $y \in [0.77465, 0.77449]$. This result is included in the invalid domain, that is $[0.77465, 0.77449] \in]0.68180, +\infty[$, and thus the verdict of the test is `KO` and the mutant is killed. If the output of the test on that mutant was inside $[0.68165, 0.68179]$, i.e. the valid range, the verdict would be `OK` and the mutant survived. By solving the output constraint of the `gen_csp_test` procedure appropriately, we ensure that the test inputs are designed to kill this mutant.

5.2 Modeling the Test CSP

Algorithm 1 is called when a mutant is not killed by the current test set. Magneto then generates a new test that is a discriminating point allowing to distinguish $\mathcal{S}$ from $\mathcal{M}$. The first step of Algorithm 1 is to generate a new CSP $\mathcal{R}$, with the procedure `gen_csp_test`, encoding a constraint that defines the distinguishing points.

In general, there are two ways of distinguishing the specification from the mutant: either Magneto finds a set of input values which are a valid solution of $\mathcal{S}$ and an invalid solution of $\mathcal{M}$, or the opposite. Let c denote the constraint of the specification and m the constraint of the mutant. The constraint $c \wedge \neg m$ encodes a value that is a valid solution of $\mathcal{S}$ and an invalid solution of $\mathcal{M}$.

For our running example specification $\mathcal{S}$ and the mutant $\mathcal{M}_1$ the full formal CSP for $c \wedge \neg m$ looks as follows:

$$\begin{aligned} &\text{Variables :} && x \in [-1.5, 1.5], \quad y \in [-\infty, \infty], \\ & && \varepsilon \in [-10^{-4}, 10^{-4}], \\ &\text{Constraint :} && y \in x^3 \cdot (1+\varepsilon) \wedge y \notin x^2 \cdot (1+\varepsilon) \end{aligned} \tag{9}$$

A more detailed analysis of the constraints shows that there are, in fact, two tests that can be generated from this constraint. We can write the constraint m with ε as a constant, as the set of solutions belonging to $m_1 \wedge m_2$, such that

$$m_1 : y \leq \max\left(x^2 \cdot (1+\varepsilon), x^2 \cdot (1-\varepsilon)\right),$$
$$m_2 : y \geq \min\left(x^2 \cdot (1+\varepsilon), x^2 \cdot (1-\varepsilon)\right).$$

Note that the ε is crucial for taking into account the finite-precision uncertainties.

Then, both of the following two constraints provide distinguishing test inputs:

$$(c \wedge (\neg m_1)), \qquad (c \wedge (\neg m_2)).$$

If we encode the constraint that the input values are a valid solution of $\mathcal{M}$ and an invalid solution of $\mathcal{S}$, i.e. the opposite case of $c \wedge \neg m$, we obtain in symmetric fashion two more CSPs:

$$((m \wedge (\neg c_1)), \qquad (m \wedge (\neg c_2)).$$

Hence, in total we can generate four different CSPs to distinguish $\mathcal{S}$ from $\mathcal{M}$. In practice, we only need one test input to kill the mutant, but the CSP solver may not be able to generate test inputs from all four CSPs. Hence, Magneto calls the CSP solver on the different CSPs until it finds a test input, and disregards the remaining ones.

The above CSP construction works for all mutations, except for `accuracy`, which is special: increasing ε means that, by definition, all valid solutions of $\mathcal{S}$ are also valid solutions of $\mathcal{M}$. For the `accuracy` mutant $\mathcal{M}_2$ (Equation 7), the test CSP $\mathcal{R}$ defines solutions of the mutant which are not solutions of the specification as follows:

$$\begin{aligned} &\text{Variables :} && x \in [-1.5, 1.5], \quad y \in [-\infty, \infty], \\ & && \varepsilon \in [-10^{-4}, 10^{-4}], \quad \varepsilon_m \in [-10^{-3}, 10^{-3}], \\ &\text{Constraint :} && y \notin x^3 \cdot (1+\varepsilon) \wedge y \in x^3 \cdot (1+\varepsilon_m). \end{aligned} \tag{10}$$

Note that for `accuracy` only the constraint $\neg c \wedge m$ makes sense, such that two tests can be generated (instead of four).

5.3 Solving the Test CSP

Solving the CSP will provide a solution in the form of a set of values $\{d_1 \in D_1, \ldots, d_n \in D_n\}$ for the variables $\{x_1, \ldots, x_n\}$, which are consistent with the constraints, that is, each variable takes a value from its definition domain so that all constraints are satisfied at the same time. Since our constraints are interval-valued, the solution of the CSP is also provided as a union of interval domains, or boxes [21], i.e. the d_i's are intervals. The CSP solver we utilize uses a branch-and-bound algorithm [16]. The algorithm starts with the full input domain. It iterates subdividing the variable domains and checking each subdomain for whether it is a solution of the given constraint. Checking of the constraints is done by evaluating them using interval arithmetic. The goal is to find a subdomain which contains only points that are solutions to the CSP; these subdomains are called `inner` boxes [13].

Subdivision is necessary for two reasons: a) the subdomain may contain points which are not solutions, or b) interval arithmetic may not be accurate enough to show that a subdomain contains only solutions. The latter happens because interval arithmetic does not keep track of correlations between variables and can thus over-approximate the true range; reducing the input domains also reduces over-approximation.

Even if no solution, i.e. `inner` box is found, the solving step eventually stops when all boxes have reached a minimum width Δ, the smallest size possible for one box.

The procedure `solve_csp` is a call to the constraint solver, to solve the constraints of $\mathcal{R}$ at the solver's working precision Δ. If an `inner` box is found, the solver can stop searching for solutions. The procedure `pick_test` then chooses values to assign to the new test t. If no test is found at the precision Δ, Δ is decreased and solving starts over. If no solution is found at the precision $\Delta_{\min}$, then the algorithm terminates without finding a new test to kill the mutant.

One of the reasons why the algorithm might not be able to find a new test comes from the fact that strict inequalities do not exist for CSP with continuous constraints. This is because continuous CSP defines the solution set as a reliable over-approximation in which no solution is lost [49]. Negated constraints of the mutants or the specification are therefore expressed with non-strict inequalities, i.e. closed intervals (this is safe, but slightly over-approximate).

Choosing Δ. The CSP solver solves the test constraint for boxes of minimum width Δ, roughly meaning that the working precision of the solver is on the order of $-\log_2(\Delta)$ bits. Choosing a value

for Δ for which the solver can find a solution is highly constraint-dependent and cannot be predicted up-front. We want to find a tradeoff between a value Δ that is small enough to find one solution of the test CSP, but not so small as to avoid creating a huge overhead on the execution time (each time Δ is decreased, the solver has to find a valid solution in a larger number of boxes).

We tackle this problem by an iterative loop, decreasing the value of Δ starting from Δ_{max}, which is set to the value of the bound of the accuracy ε. The search ends with $\Delta = \Delta_{min}$, which is a user parameter. In the worst case, the solver is called over and over again but does not find any solution.

Test Input Selection. Magneto selects the test inputs and computes the output domains with the procedure `pick_test`. The solver returns a solution of $\mathcal{R}$ in the form of boxes, that is, intervals over the inputs and output of the specification. All values in the input intervals give an output which is compliant with the constraint specification. Since test cases are composed of single-valued inputs, Magneto must pick a single value for each input.

`pick_test` chooses the test inputs as the middle value of the input intervals. To determine the valid, resp. invalid ranges, i.e. the intervals of the output for which the verdict is `OK`, resp. `KO`, `pick_test` evaluates the specification over the test input with interval arithmetic (Section 4.3).

5.4 Limitations

Equivalent Mutants. When a mutant describes a behavior which is equivalent to the solution set defined by the specification $\mathcal{S}$, then a discriminating point between $\mathcal{S}$ and the mutant cannot be found. We call this mutant an equivalent mutant [47]. Similarly, if a mutant has a solution set which is "too close" to the specification, meaning that there exist no solution at the minimum width $\Delta_{\min}$ of the CSP $\mathcal{R}$, then Magneto cannot find a test case to distinguish the mutant from the specification, and labels it as an equivalent mutant.

For example, if the specification constraint is $c : y \in (a + b)(1 + \varepsilon)$, the mutant generated from the `variable` mutation may be: $m : y \in (b + a)(1 + \varepsilon)$. Since the test CSP cannot define $\neg m$ with strict inequalities, the solver will try to find the solution $y = \max((a + b) \cdot (1 + \varepsilon), (a + b) \cdot (1 - \varepsilon))$. However this solution set is represented by single values, generally non-representable in finite precision, that do not fit into an interval of solutions. All the constraint solver can do is compute all the boxes of minimum size on the edge of the solution set of $\mathcal{S}$.

Currently, Magneto simply stops the search for a solution and leaves an error message for a tester to determine whether the mutant is equivalent or not. In future work, equivalent mutants could be detected automatically by rules over the mutations to unify CSP problems in non-equivalent cases.

Testing Program Accuracy. Our precision mutation is able to distinguish relatively large differences in errors between the specification and a mutant, such as when changing the relative error ε by an order of magnitude from 10^{-4} to 10^{-3}. It is, however, not able to distinguish arbitrary accuracy requirements.

For example, the specification

$$y = x_1 + x_2 + x_3, \text{ with}$$
$$x_1 \in [1024, 2048], \quad x_2 \in [1, 2], \quad x_3 \in [-2048, -1024]$$

can only be implemented with a relative error of 10^{-7} in single precision floating-point arithmetic if the equation is evaluated as $(x_1 + x_3) + x_2$, but not as $(x_1 + x_2) + x_3$[2]. The issue is that in order to distinguish these two expressions, we would have to have knowledge of the code and consider the precise floating-point precision and its semantics that is being used. We note that there are static analysis tools that can automatically bound roundoff errors for floating-point expressions such as the above [20, 55]. For this work, we settle on an implementation-independent approach, which is able to catch 'coarse' precision errors, such as when the entire code is using an incorrect precision.

6 IMPLEMENTATION

We have implemented the approach presented in Section 5 in the prototype tool Magneto. Magneto is written in Julia [9, 10] (Magneto needs at least Julia v1.4.2 (May 23, 2020)), which is a flexible dynamic language suitable for scientific and numerical computing, with performance comparable to those of traditional static languages. Julia offers ease and expressiveness for high-end digital computing, while ensuring reliable arithmetic using `BigFloat` (based on the arbitrary-precision library MPFR [27]) and a package providing interval arithmetic on `Float64` and `BigFloat`.

The input constraints $\mathcal{S}$ are given to Magneto as a structure consisting of

- constants, specified using `BigFloat`,
- variables, with their domain given intervals with `BigFloat` bounds
- the constraint equation as an abstract syntax tree over constants, variables, unary operations and binary operations.

The output of Magneto consists of a `Float64` assignment for the input variables (the test input), together with three `BigFloat` intervals for the output variable (for the valid and invalid verdicts). Magneto supports the operations: $+, -, \times, /, \sqrt{}$, min, max, log, exp, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh.

6.1 Modifications to Realpaver

The solver used with Magneto is `Realpaver` [32, 33], version 0.4 (2004). `Realpaver` is a constraint solver on discrete or continuous variables represented by intervals. This software allows for the modeling and solving of non-linear systems by computing reliable approximations of all the solutions. The constraints are sets of equations or inequalities on integer and real variables. The input parameters of `Realpaver` are the minimum box width Δ as the parameter *precision*, and the maximum number of boxes the solver will try to solve as the parameter *number*.

By default, `Realpaver` computes the whole solution set of a CSP, that is all `inner` and `outer` boxes. In Magneto, we are only looking for the first `inner` box. We thus modified the *search.c* source file of `Realpaver` such that the software stops after returning the first `inner` box that the solver finds. This is crucial for Magneto's

[2]The reason is that $x_1 + x_3$ does not incur a roundoff error because x_1 and x_3 have the same exponent; this is known as the Sterbenz Lemma.

Table 2: Summary of evaluation statistics

			Magneto			BB	
	# mut.	# ops	time (s)	# tests	adeq.	# tests	adeq.
min	44.0	2.0	4.0	1.0	0.69	1.0	0.69
max	1,283.0	63.0	1,083.4	9.0	1.0	8.0	1.0
mean	179.2	8.1	76.9	3.6	0.88	2.9	0.85
median	120.0	5.0	19.8	3.0	0.91	3.0	0.88

performance, since the resolution of a CSP over 100000 boxes can be very time consumming when the accuracy of the specification is high (around 10^{-5}). With that modification, the executions of `Realpaver` do not require any aditional time for non relevant boxes.

We use a single constraint solver with Magneto, but note that the interface is modular and it is thus straight-forward to integrate a different constraint solver, which may have a different performance tradeoff.

7 EXPERIMENTAL EVALUATION

In this section, we evaluate Magneto's efficiency and ability to generate adequate tests and compare it to random testing.

7.1 Setup

We conducted our experiments on an Intel Xeon E7-8857 v2 server with 4×12 cores, 3GHz clock speed and with 303GB of RAM (though Magneto does not need 300GB RAM to run), running 64bit Debian 8.3.0-6. MPFR was at version 4.0.2, based on GMP 6.1.2. All timings are wall-clock times.

Benchmarks. To the best of our knowledge, no public standard benchmark set of functional specifications for numerical software exists; usually such specifications are used for commercial proprietary safety critical software. We converted 69 benchmarks from the FPBench [19] benchmark suite to our constraint specification format. FPBench is a standard benchmark set collected from evaluations of existing static and dynamic floating-point analysis tools. Benchmarks consist mostly of straight-line (often nonlinear) arithmetic expressions with transcendental function calls that appear in scientific computing and embedded system domains.

For our evaluation, we select those 69 benchmarks that do not feature logical preconditions and loops. Most benchmarks come with interval bounds on inputs, which we use as input domains. The benchmarks do not come, however, with accuracy bounds on the results. For simplicity, we set the uniform accuracy requirement to 10^{-10} for all benchmarks. This accuracy bound is chosen such that it allows for a double-precision floating-point implementation of the specification, but not a single-precision one, in general.[3] All of our benchmarks are available at on GitHub[4].

Mutants. On average, we generate 179 mutants per benchmark, with a minimum of 44 and a maximum of 1283 mutants. When generating mutants, we avoid generating equivalent mutants that can be straight-forwardly identified with syntactic checks. Table 2 shows the min, max, mean and median of the number of mutants generated, as well as the number of arithmetic and transcendental operations in the specification expressions.

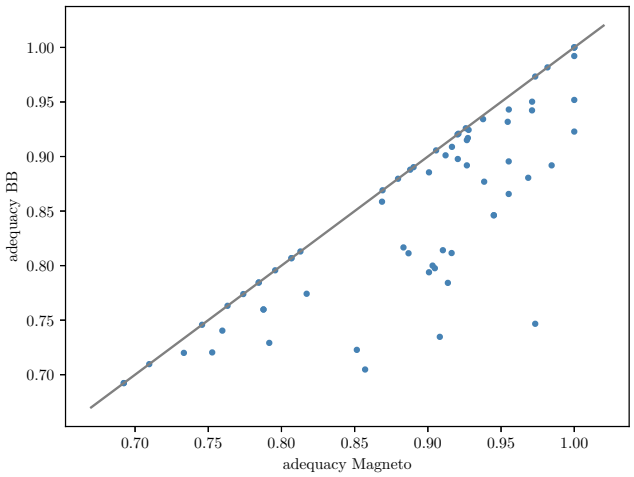


Figure 4: Adequacy of Magneto and random testing

Baseline. In the absence of an available existing tool for mutation testing of numerical specifications with interval constraints, we compare Magneto's test generation against a random, blackbox, testing baseline, as as described in Section 4.3. We denote this baseline as 'BB'.

The performance of random testing clearly depends on the time limit used for generating new tests. To allow for a reasonably fair comparison, we first run Magneto and measure its execution time. We then run random testing with a time limit set to Magneto's time, i.e. a different time limit for each benchmark. We limit Magneto's execution time by setting a timeout of 10s on each CSP call.

7.2 Results

Adequacy. We compare the adequacy scores of Magneto and black-box testing (BB) in Figure 4. When a dot is below the gray line, Magneto's adequacy score was higher than blackbox's for a given benchmark. If the dot is on the line, the adequacy is equal. Table 2 furthermore provides the minimum, maximum, mean and median adequacy scores.

Overall, we observe that Magneto outperforms BB over our benchmark set. Magneto always kills at least as many mutants as random testing, and for 65% of the benchmarks, Magneto has a higher mutation score than BB. This is also reflected in the higher average and median adequacy scores.

When the adequacy score of BB is lower than Magneto's, random testing failed to kill mutants which could only be killed with low probability inputs. When Magneto does not kill a mutant, it may be because the timeout was too small or because the mutant was equivalent. We manually checked the remaining live mutants; about half of them were indeed equivalent mutants (that could not be easily syntactically identified), whereas for the other half the time limit or precision of the CSP solver was not sufficient.

Nevertheless, our evaluation shows that Magneto is able to generate overall adequate test sets.

[3] The machine epsilon for single and double precision is 5.96e-08 and 1.11e-16, resp.
[4] https://github.com/clothildejeangoudoux/Magneto

Table 3: Total number of mutants per mutation type over all the benchmarks

type	# mutants	# killed BB	# killed Magneto
accuracy	544	422	498
bounds	270	122	196
constant	240	227	228
variable	404	402	402
unary	938	783	795
binary	2427	2401	2404
add	7035	5996	6306
del	245	244	244

Types of Mutants Killed. Table 3 shows the total number of mutants generated per category over the entire benchmark set, as well as the number of mutants killed by Magneto and random testing. For every mutation category, the test sets generated by Magneto kill more mutants than BB, which is consistent with the adequacy scores. Magneto was particularly better in killing accuracy (78% vs 92%) and bounds (45% vs 73%) and add (85% vs 90%) mutants. Those mutants represent a small fraction of the total number of mutants, so the overall adequacy difference may look small, but generating adequate tests to kill them takes most of the computation effort.

Examples. We discuss three benchmarks in more detail. `nonlin1` ($y = z/(z+c)$) is a small benchmark with 2 operations and 65 mutants. Both Magneto and BB perform badly (adequacy score of 0.69), because the input is defined over a large floating-point domain that is divided by almost itself (which is numerically instable). Hence BB has low chances to find tests producing a clear `KO` verdict for some mutants, and Magneto's CSP solver times out.

`doppler2` ($y = ((-t1)*v)/((t1+u)*(t1+u))$with $t_1 = 1657/5 + 3/5 * T$) is a medium sized benchmark with 259 mutants. Magneto scored 0.98 while BB's adequacy was 0.89. BB could not kill `bounds` and some `add` mutants, while Magneto generated only one adequate test that killed 98% of the mutants in 44 seconds. The 4 `add` mutants that Magneto could not kill were equivalent mutants.

Finally, `i6` ($y = \sin(x1 + x2)$) is small benchmark with 75 mutants. Magneto scored 0.97 while BB's adequacy was only 0.74. Here, BB could not kill the `accuracy` and `bounds` mutants, while Magneto generated only one adequate test that killed 97% of the mutants in only 8 seconds.

Runtime. Figure 5 shows Magneto's runtime. The average runtime of Magneto over all benchmarks is 77 seconds, over only the 7 most complex ones the average runtime is higher at 200 seconds. In 70% of the cases, Magneto's runtime is lower than 50 seconds. The slowest execution rounds up to 18 minutes (not shown in Figure 5 for readability reasons), while the fastest one took 4 seconds.

Most of the runtime of Magneto is spent on CSP solving. We observed that the solver usually returns an output almost immediately. This is, however, not the case for the equivalent mutants detection problem. In order to tag a mutant as equivalent, Magneto has to repeatedly call the CSP solver with decreasing values of Δ,

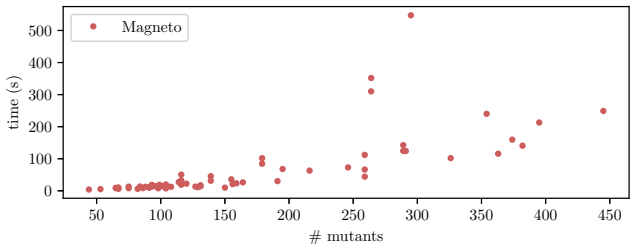


Figure 5: Magneto's runtime on different benchmarks

until Δ_{min} is reached. In that case, either `RealPaver` does not find any solution or reach a timeout of 10 seconds.

We conclude from our experiments that Magneto is globally reliable with the currently implemented mutations. It makes it possible to generate relevant tests in a reasonable time (less than two minutes on average), and produces test sets that are more adequate to the specification than random testing.

8 RELATED WORK

Metamorphic testing is a property-based software testing used to determine reliable test oracles [14] by using constraint logic programming to generate a model of the program [30]. New test cases aiming at uncovering specific errors are generated by injecting faults. Magneto differs from metamorphic testing in two aspects. First, by generating the model from the code, metamorphic testing may introduce complexity by focusing on how the result is computed rather than the functionality the result should implement. Secondly, Magneto takes into account the accuracy of the specification as part of the test oracle. Hence when computing with floating-point numbers, the slight but acceptable variation in the test results must be compared to a heuristic test oracle [46, 54]. The complexity of the search space and the lack of reliable accuracy constraints prevent us from using metamorphic testing over numerical specification.

Model-Based Testing refers to a range of testing methods, from test oracle generation, to reliable test evaluation, for all types of programs [34, 57]. Each technique is inherently tied to the chosen model. Popular in industry for critical software development, SCADE [58] and Matlab/Simulink [53] use model-checking [26] approaches to generate tests. The problem with using those techniques with certified software is that the code is automatically generated from those models. Tests generated from these models cannot be used in the avionics software certification process according to the DO-178C standard [52].

Mutation testing has been applied to both program source code and specification. For the latter, the faults are injected into the program specification while never looking at the program source code. Mutation testing over formal specifications has been studied for predicate calculus [12], calculus specification [5] and algebraic specification [59]. However, none of those types of formal specification allow the user to specify the behavior of approximations of numerical systems. A numerical test generation technique over n-dimensional polynomial models [40] shows that it is significantly

faster at killing mutants than iterative random testing. This work, however, does not tackle nonlinear models.

Equivalent mutant detection [44] can be tackled using an co-evolutionary approach [4], or constraint representation of the program and the mutants [47]. Constraint solving over polynomial systems [35] could resolve this, but they do not support transcendental functions.

Higher order mutation testing [36] has been studied as a way to find subtle faults that classical mutation testing can scarcely emulate. Higher order mutations are useful to capture more faults with fewer mutants, in an effort to reduce the computation cost of mutation testing techniques. In [48], the authors show in their experiments that using second order mutants reduces the test effort by 50% on programs using integers.

Static analyses which compute guaranteed upper bounds on floating-point roundoff errors [20, 22, 31, 39, 43, 55] are complementary to our approach in that they analyze the actual finite precision implementation. They, in general, over-approximate the true errors by about an order of magnitude, for expressions similar in size to our benchmarks.

Constraint programming has also been studied for the verification of floating-point programs [3]. Here, constraint programming represents a model of the code, which is used either to verify properties on the code [51], or to generate test cases inside suspicious intervals [18]. For the latter technique, user program annotations link the constraints and the suspicious program executions. We solve similar problems without looking at the code.

Testing techniques targeting floating-point arithmetic use, for instance, black-box techniques and a shadow execution with a higher precision version of the code to find inputs that cause large roundoff errors [8, 15, 62]. Recent white-box testing approaches have focused on detecting overflows [28] or large roundoff errors [63], using mathematical optimization and condition numbers, respectively. Symbolic execution [7, 38] has also been used to find bugs and overflows in floating-point code. While these techniques are useful for debugging the accuracy of the code, they are not applicable to check its compliance with a specification.

9 CONCLUSION

We have shown how to systematically generate tests from numerical specifications with accuracy requirements using interval constraint programming. Our approach outperforms a random testing baseline and produces adequate test sets, catching many common programming bugs. An interesting perspective for future work is to improve the detection of equivalent mutants and to take into account a high-level semantics of floating-point arithmetic, in order to kill more mutants.

REFERENCES

[1] 2006. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence, Vol. 2. Elsevier. http://www.sciencedirect.com/science/bookseries/15746526/2

[2] 2008. *IEEE Standard for Floating-Point Arithmetic*. https://doi.org/10.1109/IEEESTD.2008.4610935

[3] Carlos Acosta, Martine Ceberio, and Christian Servin. 2008. A Constraint-Based Approach to Verification of Programs with Floating-Point Numbers. In *Software Engineering Research & Practice (SERP)*.

[4] Konstantinos Adamopoulos, Mark Harman, and Robert Hierons. 2004. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In *Genetic and Evolutionary Computation (GECCO)*. https://doi.org/10.1007/978-3-540-24855-2_155

[5] Bernhard Aichernig. 2002. Mutation Testing in the Refinement Calculus. *Electr. Notes Theor. Comput. Sci.* 70 (2002). https://doi.org/10.1016/S1571-0661(05)82561-7

[6] Roberto Amadini. 2015. *Portfolio Approaches in Constraint Programming*. Ph.D. Dissertation. University of Bologna, Italy. https://tel.archives-ouvertes.fr/tel-01227582

[7] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2429069.2429133

[8] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2345156.2254118

[9] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. [n.d.]. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* ([n. d.]). https://doi.org/10.1137/141000671

[10] Jeff Bezanzon, Stefan Karpinski, Viral Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. In *Lang.NEXT*.

[11] Timothy Budd and Dana Angluin. 1982. Two Notions of Correctness and Their Relation to Testing. *Acta Inf.* 18 (1982). https://doi.org/10.1007/BF00625279

[12] Timothy A. Budd and Ajei S. Gopal. 1985. Program Testing by Specification Mutation. *Comput. Lang.* 10, 1 (1985). https://doi.org/10.1016/0096-0551(85)90011-6

[13] Gilles Chabert and Luc Jaulin. 2009. Contractor programming. *Artificial Intelligence* 173, 11 (2009). https://doi.org/10.1016/j.artint.2009.03.002

[14] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 2015. Metamorphic Testing: A New Approach for Generating Next Test Cases. arXiv:2002.12543 [cs.SE]

[15] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/2555243.2555265

[16] Jens Clausen. 1999. Branch and Bound Algorithms – Principles And Examples. https://imada.sdu.dk/~jbj/DM85/TSPtext.pdf

[17] Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software traceability: Trends and future directions. *Future of Software Engineering, FOSE 2014 - Proceedings* (2014). https://doi.org/10.1145/2593882.2593891

[18] Hélène Collavizza, Claude Michel, Olivier Ponsini, and Michel Rueher. 2014. Generating Test Cases Inside Suspicious Intervals for Floating-point Number Programs. In *Constraints in Software Testing, Verification, and Analysis (CSTVA)*. https://doi.org/10.1145/2593735.2593737

[19] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. [n.d.]. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. ([n. d.]). https://doi.org/10.1007/978-3-319-54292-8_6

[20] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-319-89960-2_15

[21] Hend Dawood. 2011. *Theories of Interval Arithmetic: Mathematical Foundations and Applications*. LAP Lambert Academic Publishing.

[22] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. 2006. Assisted Verification of Elementary Functions Using Gappa. In *ACM Symposium on Applied Computing*. https://doi.org/10.1007/978-3-642-18275-4_17

[23] Pedro Delgado-Pérez, Ana B. Sánchez, Sergio Segura, and Inmaculada Medina-Bulo. 2020. Performance mutation testing. *Software Testing, Verification and Reliability* (2020). https://doi.org/10.1002/stvr.1728

[24] R. A. DeMillo and A. J. Offutt. 1991. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 17, 9 (1991). https://doi.org/10.1109/32.92910

[25] M. Ellims, D. Ince, and M. Petre. 2007. The Csaw C Mutation Tool: Initial Results. In *Testing: Academic and Industrial Conference Practice and Research Techniques)*. https://doi.org/10.1109/TAIC.PART.2007.28

[26] Eduard P. Enoiu, Adnan Čaušević, Thomas J. Ostrand, Elaine J. Weyuker, et al. 2016. Automated Test Generation Using Model Checking: An Industrial Evaluation. *Int. J. Softw. Tools Technol. Transf.* 18, 3 (2016). https://doi.org/10.1007/s10009-014-0355-9

[27] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (2007). https://doi.org/10.1145/1236463.1236468

[28] Zhoulai Fu and Zhendong Su. 2019. Effective floating-point analysis via weak-distance minimization. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314632

[29] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.* 23, 1 (1991). https://doi.org/10.1145/103162.103163

[30] Arnaud Gotlieb and Bernard Botella. 2003. Automated Metamorphic Testing. In *27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems, 3-6 November 2003, Dallas, TX, USA, Proceedings*. https://doi.org/10.1109/CMPSAC.2003.1245319

[31] E. Goubault and S. Putot. 2011. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

[32] Laurent Granvilliers. 2004. *RealPaver User's Manual Solving Nonlinear Constraints by Interval Computations*.

[33] Laurent Granvilliers and Frédéric Benhamou. 2006. Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software* 32, 1 (2006). https://doi.org/10.1145/1132973.1132980

[34] Havva Gulay Gurbuz and Bedir Tekinerdogan. 2017. Model-based testing for software safety: a systematic mapping study. *Software Quality Journal* (2017). https://doi.org/10.1007/s11219-017-9386-2

[35] Chris Jefferson, Peter Jeavons, Martin Green, and M. Dongen. 2013. Representing and solving finite-domain constraint problems using systems of polynomials. *Annals of Mathematics and Artificial Intelligence* 67 (2013). https://doi.org/10.1007/s10472-013-9365-7

[36] Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Inf. Softw. Technol.* 51, 10 (2009). https://doi.org/10.1016/j.infsof.2009.04.016

[37] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011). https://doi.org/10.1109/TSE.2010.62

[38] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. 2017. Floating-Point Symbolic Execution: A Case Study in N-Version Programming. In *ASE*. https://doi.org/10.1109/ASE.2017.8115670

[39] Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.* 43, 4 (2017). https://doi.org/10.1145/3015465

[40] Karl Meinke and Fei Niu. 2010. A Learning-Based Approach to Unit Testing of Numerical Software. In *Testing Software and Systems*.

[41] R.E. Moore. 1966. *Interval Analysis*. Prentice-Hall.

[42] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. 2009. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics.

[43] Mariano Moscato, Laura Titolo, Aaron Dutle, and Cesar Muñoz. 2017. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *SAFECOMP*. https://doi.org/10.1007/978-3-319-66266-4_14

[44] Elfurjani Mresa and Leonardo Bottaci. 1999. Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. *Softw. Test., Verif. Reliab.* 9 (1999). https://doi.org/10.1002/(SICI)1099-1689(199912)9:43.0.CO;2-X

[45] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, et al. 2018. *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhäuser Boston. 632 pages.

[46] Christian Murphy, Kuang Shen, and Gail Kaiser. 2009. Automatic System Testing of Programs without Test Oracles. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/1572272.1572295

[47] Simona Nica and Franz Wotawa. 2012. Using Constraints for Equivalent Mutant Detection. In *Workshop on Formal Methods in the Development of Software (WS-FMDS)*.

[48] Macario Polo, Mario Piattini, and Ignacio García Rodríguez de Guzmán. 2009. Decreasing the cost of mutation testing with second-order mutants. *Softw. Test. Verification Reliab.* 19 (2009).

[49] Olivier Ponsini, Claude Michel, and Michel Rueher. 2012. Refining Abstract Interpretation Based Value Analysis with Constraint Programming Techniques. In *Principles and Practice of Constraint Programming (CP)*. https://doi.org/10.1007/978-3-642-33558-7_43

[50] Olivier Ponsini, Claude Michel, and Michel Rueher. 2016. Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Automated Software Engineering* (2016). https://doi.org/10.1007/s10515-014-0154-2

[51] Olivier Ponsini, Claude Michel, and Michel Rueher. 2016. Verifying Floating-point Programs with Constraint Programming and Abstract Interpretation Techniques. *Automated Software Engg.* 23, 2 (2016). https://doi.org/10.1007/s10515-014-0154-2

[52] RTCA Special Committee 205 and EUROCAE Working Group 71. 2011. *DO–178C, ED–12C, Software Considerations in Airborne Systems and Equipment Certification*. Technical Report. RTCA, Inc.

[53] Artur Schmidt, Umut Durak, Christoph Rasch, and Thorsten Pawletta. 2015. Model-Based Testing Approach for MATLAB/Simulink using System Entity Structure and Experimental Frames. In *Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*.

[54] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016). https://doi.org/10.1109/TSE.2016.2532875

[55] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. 2015. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *Formal Methods (FM)*. https://doi.org/10.1145/3230733

[56] Ashish Tiwari and Patrick Lincoln. 2016. A search-based procedure for nonlinear real arithmetic. *Formal Methods in System Design* 48 (2016). https://doi.org/10.1007/s10703-016-0245-8

[57] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-Based Testing Approaches. *Softw. Test. Verif. Reliab.* 22, 5 (2012). https://doi.org/10.1002/stvr.456

[58] Virginia Papailiopoulou, Besnik Seljimi, and Ioannis Parissis. 2011. Automatic Testing of LUSTRE/SCADE Programs. In *Model-based testing for embedded systems*. CRC Press.

[59] M. R. Woodward. 1992. OBJTEST: an experimental testing tool for algebraic specifications. In *IEE Colloquium on Automating Formal Methods for Computer Assisted Prototying,*.

[60] Tao Xie, Wolfgang Mueller, and Florian Letombe. 2012. Mutation-analysis driven functional verification of a soft microprocessor. In *IEEE 25th International SOC Conference (SOCC)*. https://doi.org/10.1109/SOCC.2012.6398362

[61] Randy Yates. 2013 (accessed May 9, 2020). *Fixed-Point Arithmetic: An Introduction*. http://www.digitalsignallabs.com/fp.pdf

[62] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2015.70

[63] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2020. Detecting floating-point errors via atomic conditions. *PACMPL* 4, POPL (2020). https://doi.org/10.1145/3371128