Interval Constraint-Based Mutation Testing of Numerical Specifications

Anonymous Author(s)

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.

ACM Reference Format:

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. 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].

Given a target specification, our algorithm first generates a number of mutants from three categories of faults: robustness, functional behavior, and accuracy. These, respectively, test whether the code supports the correct input range, whether it implements the correct arithmetic expression, and whether it is implemented with sufficient precision.

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 our 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 increase the probability of generating such a test input, 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.
Figure 1: Mutation Testing for Automated Test Generation

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 and is particularly useful in the strict certification process of safety-critical software that prohibits the use of random testing [52].

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.

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), which we will release as open-source, 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

\[ r = 5 \cdot \frac{x_1 - x_2}{x_2} \]  

(1)

defines the extracted air flow of a helicopter engine, and will serve as the specification given to Magneto that we will denote by \( S_r \).

This specification is real valued, i.e. \( x_2, r \in \mathbb{R} \), 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:

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

(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 \( S_r \) 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 \( M \), Magneto injects a fault, e.g. a syntactic change that corresponds to a possible programming error, into the specification \( S \). Magneto then iteratively generates test inputs that distinguish between \( S \) and the mutants. We say that the verdict of a test is KO, if a test input found an injected bug in a mutant (\( \text{killed} \) 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 \( M_r \):

\[ r_M = 5 \cdot \frac{x_1 + x_3}{x_2} \]  

(3)

Initially, the test set \( T \) is empty, so that the mutant \( M_r \) 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, \( r_M \in [5.49999 \cdot 10^{-7}, 5.00001 \cdot 10^{-8}] \). In order to distinguish between \( S_r \) and \( M_r \), Magneto thus has to identify an input, which is in the result interval of \( M_r \), but not in \( S_r \), 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:

\[ r \in 5 \cdot \frac{x_1 - x_3}{x_2} \land r \notin 5 \cdot \frac{x_1 + x_3}{x_2} \]  

(4)

This constraint is interval-valued, i.e. the input variables \( x_1 \) are bounded by intervals. With some abuse of notation, we write \( r \in f(x_1, x_2, x_3, x_4) \) 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 \( S_r \) results in \( r = -50000.5 \), but when executed on \( M_r \), 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 KO 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 $\varepsilon_M = 10^{-38}$ (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 $M$, but not by $S$ since the valid range of $S$ is by necessity, included in the valid range of $M$. 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 K0 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 $\tilde{x}$:

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

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

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

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 $0 \leq |m| < 2^p$. 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 $e_{\max} = -e_{\min} + 1 = 2^{11-1} - 1$. 

---

**Figure 2: Different verdict scenarios**

<table>
<thead>
<tr>
<th>Verdict</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>Test passes</td>
</tr>
<tr>
<td>KO</td>
<td>Test fails</td>
</tr>
<tr>
<td>UNCERTAIN</td>
<td>Test undecided</td>
</tr>
</tbody>
</table>

![Figure 2: Different verdict scenarios](image-url)
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: \( X = [\underline{x}, \overline{x}] := \{ x \mid \underline{x} \leq x \leq \overline{x} \} \). Binary arithmetic operations over intervals \( X \) and \( Y \) soundly enclose all possible results of the operation assuming that the input arguments are in \( X \) and \( Y \):

\[
X \circ Y \supseteq \left\{ x \circ y \mid x \in X, y \in Y \right\}
\]

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 \( \pi \). 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 \( S \) as running example to illustrate our explanations:

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

#### 4.1 Constraints as Input Specification

The input to our mutation testing algorithm is a formal specification \( S \). 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.

---

**Table 1: Set of mutations considered in Magneto**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accuracy bounds</td>
<td>decrease the accuracy by increasing the error bound ( \varepsilon )</td>
</tr>
<tr>
<td>constant variable</td>
<td>replace the value of a constant variable with a new value</td>
</tr>
<tr>
<td>unary variable</td>
<td>replacement of an unary operator with another ( (\sqrt{\cdot}, \sin, \cos, \tan, \exp, \log) )</td>
</tr>
<tr>
<td>binary variable</td>
<td>replacement of an binary operator with another ( (+, -, \times, \div, \max, \min) )</td>
</tr>
<tr>
<td>add</td>
<td>add an operator</td>
</tr>
<tr>
<td>del</td>
<td>delete an operator</td>
</tr>
</tbody>
</table>

---

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

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

For an input specification \( S \), 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 \( \varepsilon \) 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:

\[
\text{Variables : } \quad x \in [-1.5, 1.5], \quad y \in [-\infty, \infty], \quad \varepsilon \in [-10^{-4}, 10^{-4}]
\]

\[
\text{Constraint : } \quad c : y \in x^3 \cdot (1 + \varepsilon)
\]

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 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).

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

\[
(M_1) \quad y \in x^3 \cdot (1 + \varepsilon_m), \quad |\varepsilon_m| \leq 10^{-3}, \quad x \in [-1.5, 1.5] 
\]

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

\[
(M_2) \quad y \in x^3 \cdot (1 + \varepsilon), \quad |\varepsilon| \leq 10^{-4}, \quad x_m \in [-1.5, 1.6] 
\]

The mutation modifies the bounds of the variable \( x \) to form the variable \( x_m \).

### 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 mutation 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 **K0** if the mutant’s valid range is included in the valid 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.

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.

![Figure 3: Accuracy mutant comparison with test oracle](image)

**Example.** For the requirement \( S \) (Equation 6), two values \( y_1 \) and \( y_2 \) are computed from the input input \( x \) with interval arithmetic, such that \( y_1 = x^3 \cdot (1 + \varepsilon) = [y_1, y_1'] \) and \( y_2 = x^3 \cdot (1 - \varepsilon) = [y_2, y_2'] \). 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 \([y_1, y_2]\), and the invalid range \([\mathbb{I}, \infty) \cup [y_2, +\infty[\). **Accuracy Mutation.** In the case of 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 mutant (bottom line) always includes the valid range of the specification (top line). Effectively, generating tests that are guaranteed to capture accuracy errors requires knowledge of the actual code. Instead, we aim at finding test inputs for which the overlap between the mutant valid range and the specification invalid range is ‘big’. Intuitively, this means that the probability that the test input discovers an accuracy violation is also large. We consider an overlap big enough when the difference between the boundaries of the valid and invalid range is larger than three orders of magnitude of the specified relative error \( \varepsilon \), i.e. if \( \varepsilon = 10^{-7} \), the difference must be larger than \( 10^{-4} \).

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 an accuracy issue. 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 \( x_1 = 49 \cdot 10^{-5}, x_2 = 5 \cdot 10^{-7}, x_3 = 10^{-3}, x_4 = 2 \cdot 10^{-1} \). For those inputs Magneto defines the valid output in the interval \( r_S \in ]-100.0000099, -99.9999900[ \) and the invalid output in \( r_M \in ]-\infty, 100.00000799[ \cup ]-99.9999100, +\infty[ \), whereas the valid range of the mutant is \( r_M \in ]-100.00311, -99.9983985[ \). Figure 3 shows these ranges at the top for the specification and below for the mutant. Since the interval \( ]-100.00311, -100.00000799[ \) represent the overlapping range, and \( ]-100.00311 + 100.00000799[ \) > \( 10^{-4} \), 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 straightforward 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 $S$, 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 $S$ and one mutant $M$, 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 $R$ (gen_csp_test) that defines the set of values that are solutions of $S$ but not solutions of $M$ (Section 5.2).

The procedure solve_csp calls a CSP solver in order to solve $R$, 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 $\Delta$, given as a parameter to the solver. If no solution was found, $\Delta$ is decreased, and the solving starts again, but will now take potentially longer as with a smaller $\Delta$, the solver has to perform more subdivisions. Magneto repeats this loop until $\Delta = \Delta_{\text{min}}$, for a predetermined value of minimum interval width, which we discuss further in Section 5.3.

If a solution is found that solves $R$, 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 $\Delta_{\text{min}}$ without finding a solution (line 13), the mutant $M$ is equivalent or too similar to the specification $S$ 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 $S$ (Equation 6), we obtain the following mutant:

$$(M_1) \quad y \in x^2 \cdot (1 + \varepsilon), \quad |\varepsilon| \leq 10^{-4}, \quad x \in [-1.5, 1.5] \quad (8)$$

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

$$\left[ y \in x^3 \cdot (1 + \varepsilon) \right] \land \left[ y \notin x^2 \cdot (1 + \varepsilon) \right],$$

$$|\varepsilon| \leq 10^{-4}, \quad x \in [-1.5, 1.5].$$

With $\Delta = \Delta_S = 10^{-4}$, 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 $y = [0.681645, 0.68165]$ and $\overline{y} = [0.68179, 0.68180]$. Thus, the test input and oracle generated for the mutant in Equation 8 is:

$$x = 0.8801,$$

$$y \in [0.68165, 0.68179] \rightarrow \text{OK},$$

$$y \in ]-\infty, 0.68165[ \cup ]0.68180, +\infty[ \rightarrow \text{KO}.$$
Note that the \( \epsilon \) is crucial for taking into account the finite-precision uncertainties.

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

\[
(\epsilon \land \neg m_1), \quad (\epsilon \land \neg m_2).
\]

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

\[
(m \land \neg c_1), \quad (m \land \neg c_2).
\]

Hence, in total we can generate four different CSPs to distinguish \( S \) from \( M \). 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: decreasing \( \epsilon \) means that, by definition, all valid solutions of \( S \) are also valid solutions of \( M \). For the accuracy mutant \( M_\epsilon \) (Equation 7), the test CSP \( R \) defines solutions of the mutant which are not solutions of the specification as follows:

\[
\text{Variables:} \quad x \in [-1.5, 1.5], \quad y \in [-\infty, \infty], \\
\epsilon \in [-10^{-4}, 10^{-4}], \quad \epsilon_m \in [-10^{-3}, 10^{-3}].
\]

\[
\text{Constraint:} \quad a \not\in x^3 \cdot (1 + \epsilon) \land y \in x^3 \cdot (1 + \epsilon_m).
\]

Note that for accuracy only the constraint \( \neg c \land \neg m \) 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 \( \Delta \), the smallest size possible for one box.

The procedure \texttt{solve.csp} is a call to the constraint solver, to solve the constraints of \( R \) at the solver’s working precision \( \Delta \). If an inner box is found, the solver can stop searching for solutions. The procedure \texttt{pick.test} then chooses values to assign to the new test \( t \). If no test is found at the precision \( \Delta \), \( \Delta \) is decreased and solving starts over. Is no solution is found at the precision \( \Delta_{\text{min}} \), 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 \( \Delta \). The CSP solver solves the test constraint for boxes of minimum width \( \Delta \), roughly meaning that the working precision of the solver is on the order of \( -\log_2(\Delta) \) bits. Choosing a value for \( \Delta \) 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 \( \Delta \) 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 \( \Delta \) 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 \( \Delta \) starting from \( \Delta_{\text{max}} \), which is set to the value of the bound of the accuracy \( \epsilon \). The search ends with \( \Delta = \Delta_{\text{min}} \), 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 \texttt{pick.test}. The solver returns a solution of \( R \) 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.

\texttt{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, \texttt{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 \( S \), then a discriminating point between \( S \) 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 \( \Delta_{\text{min}} \) of the CSP \( R \), 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 \( c : y \in (a + b) \cdot (1 + \epsilon) \), the mutant generated from the variable mutation may be \( m : y \in (b + a) \cdot (1 + \epsilon) \). Since the test CSP cannot define \( \neg m \) with strict inequalities, the solver will try to find the solution \( y = \max((a + b) \cdot (1 + \epsilon), (a + b) \cdot (1 - \epsilon)) \). 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 $S$.

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 $\varepsilon$ 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

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

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 know the source code 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] (Mag-netoneeds 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 $S$ 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: $+, -, \times, /, \sqrt{\cdot}$, 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 $\Delta$ 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 performance, since the resolution of a CSP over 100000 boxes can be very time consuming when the accuracy of the specification is high (around $10^{-3}$). With that modification, the executions of Realpaver do not require any additional 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 of 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 our anonymized repository\(^4\).

**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

\(^3\)The machine epsilon for single and double precision is 5.96e-08 and 1.11e-16, resp.

\(^4\)https://anonymous.4open.science/r/4916a0c-25c3-44df-9ee7-bb7b80b72b7e/
We then run random testing with a time limit set to Magneto’s time. Table 2 furthermore provides the minimum, maximum, mean and average 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.

### 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.

Baselines. 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.

### 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.

### Table 2: Summary of evaluation statistics

<table>
<thead>
<tr>
<th></th>
<th># mut.</th>
<th># ops</th>
<th>time (s)</th>
<th># tests</th>
<th>adec.</th>
<th># tests</th>
<th>adec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>44.0</td>
<td>2.0</td>
<td>4.0</td>
<td>1.0</td>
<td>0.69</td>
<td>1.0</td>
<td>0.69</td>
</tr>
<tr>
<td>max</td>
<td>1,283.0</td>
<td>63.0</td>
<td>1,083.4</td>
<td>9.0</td>
<td>1.0</td>
<td>8.0</td>
<td>1.0</td>
</tr>
<tr>
<td>mean</td>
<td>179.2</td>
<td>8.1</td>
<td>76.9</td>
<td>3.6</td>
<td>0.88</td>
<td>2.9</td>
<td>0.85</td>
</tr>
<tr>
<td>median</td>
<td>120.0</td>
<td>5.0</td>
<td>19.8</td>
<td>3.0</td>
<td>0.91</td>
<td>3.0</td>
<td>0.88</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>type</th>
<th># mutants</th>
<th># killed BB</th>
<th># killed Magneto</th>
</tr>
</thead>
<tbody>
<tr>
<td>accuracy</td>
<td>944</td>
<td>422</td>
<td>498</td>
</tr>
<tr>
<td>bounds</td>
<td>270</td>
<td>122</td>
<td>196</td>
</tr>
<tr>
<td>constant</td>
<td>240</td>
<td>227</td>
<td>228</td>
</tr>
<tr>
<td>variable</td>
<td>404</td>
<td>402</td>
<td>402</td>
</tr>
<tr>
<td>unary</td>
<td>938</td>
<td>783</td>
<td>795</td>
</tr>
<tr>
<td>binary</td>
<td>2427</td>
<td>2401</td>
<td>2404</td>
</tr>
<tr>
<td>add</td>
<td>7035</td>
<td>5996</td>
<td>6306</td>
</tr>
<tr>
<td>del</td>
<td>245</td>
<td>244</td>
<td>244</td>
</tr>
</tbody>
</table>

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.

**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. We see that 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.

**Examples.** We study in details three benchmarks: nonlin1 for which all methods perform badly (mutation score of 0.69), and doppler2 and i6 for which Magneto performs better than random testing.

In the first case, nonlin1 \((y = z/(z + c))\) is a small benchmark with 2 operations and 65 mutants. The bad performances of Magneto and BB over this benchmark come from the fact that the input, defined over a large floating-point domain, is devided by almost itself. Hence BB has low chances to find tests producing a clear KO verdict for some mutants, and Magneto’s CSP solver times out.

In the second case, doppler2 \((y = ((-t1) + v)/(t1 + u) + (t1 + u)/((t1 + u) + (t1 + u)) with t1 = 1657/5 + 3/1 + T)\) is a medium sized benchmark with 259 mutants. Magneto scored 0.98 while BB’s adequacy was 0.89. The difference in mutation scores comes from the fact that BB could not kill bounds and some add mutants with 2 tests, 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 0.74. The difference in mutation scores comes from the fact that BB could not kill the accuracy and bounds mutants with 3 tests, while Magneto generated only one adequate test that killed 97% of the mutants in only 8 seconds.
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 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


Mutation Testing of Numerical Specifications

ISSSTA 2021, 12-16 July, 2021, Aarhus, Denmark

11
