Sound Compilation of Reals

Eva Darulova
EPFL
eva.darulova@epfl.ch

Viktor Kuncak
EPFL
viktor.kuncak@epfl.ch

Abstract

Writing accurate numerical software is hard because of many sources of unavoidable uncertainties, including finite numerical precision of implementations. We present a programming model where the user writes a program in a real-valued implementation and specification language that explicitly includes different types of uncertainties. We then present a compilation algorithm that generates a finite-precision implementation that is guaranteed to meet the desired precision with respect to real numbers. Our compilation performs a number of verification steps for different candidate precisions. It generates verification conditions that treat all sources of uncertainties in a unified way and encode reasoning about finite-precision roundoff errors into reasoning about real numbers. Such verification conditions can be used as a standardized format for verifying the precision and the correctness of numerical programs. Due to their non-linear nature, precise reasoning about these verification conditions remains difficult and cannot be handled using state-of-the art SMT solvers alone. We therefore propose a new procedure that combines exact SMT solving over reals with approximate and sound affine and interval arithmetic. We show that this approach overcomes scalability limitations of SMT solvers while providing improved precision over affine and interval arithmetic. Our implementation gives promising results on several numerical models, including dynamical systems, transcendental functions, and controller implementations.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification

Keywords roundoff error; floating-point arithmetic; fixed-point arithmetic; verification; compilation; embedded systems; numerical approximation; scientific computing; sensitivity analysis

1. Introduction

Writing numerical programs is difficult, in part because the programmer needs to deal not only with the correctness of the algorithm but also with different forms of uncertainties. Program inputs may not be exactly known because they come from physical experiments or were measured by an embedded sensor. The computation itself suffers from roundoff errors at each step, because of the use of finite-precision arithmetic. In addition, resources like energy may be scarce so that only a certain number of bits are available for the numerical data type.

At the same time, the computed results in these domains can have far-reaching consequences if used to control, for example, a vehicle or a nuclear power plant. It is therefore becoming increasingly urgent to develop tools that improve confidence in numerical code [38]. One of the first challenges in doing this is that most of our automated reasoning tools work with real arithmetic, whereas the code is implemented with finite-precision. Many current approaches to verify numerical programs start with the finite-precision implementation and then try to verify the absence of (run-time) errors. Not only are such verification results specific to a given representation of numbers, but the absence of run-time errors does not guarantee that program behavior matches the desired specification expressed using real numbers. Fundamentally, the source code semantics is currently expressed in terms of low-level data types such as floating-points. This is problematic not only for developers but also for compiler optimizations, because, e.g., the associativity law is unsound with respect to such source code semantics.

In this paper we advocate a natural but ambitious alternative: source code programs should be expressed in terms of mathematical real numbers. In our system, the programmer writes a program using a **Real** data type, states the desired postconditions, and specifies explicitly the uncertainties as well as the desired target precision. It is then up to our trustworthy compiler to check, taking into account all uncertainties and their propagation, that the desired precision can be soundly realized in a finite-precision implementation. If so, the compiler chooses and emits one such implementation, selecting from a range of (software or hardware) floating-point or fixed-point arithmetic representations.

A key question that such a compiler needs to answer is whether a given finite-precision representation remains close enough to an ideal implementation in terms of real numbers. To answer this question, we present a method to generate verification conditions that encode reasoning about finite-precision roundoff errors into reasoning about real numbers. Our verification conditions explicitly model the *ideal* program without external uncertainties and roundoffs, the *actual* program, which is executed in finite precision with possibly noisy inputs, and the relationship between the two. Solving such verification conditions is one of the key tasks of a sound compiler for reals.

Our approach is parametric in the bitwidth of the representations and can thus be used on different platforms, from embedded controllers without floating-point units (where fixed-point implementations are needed), to platforms that expose high-precision floating-point arithmetic in their instruction set architecture. Using libraries we can also emit code that uses precision twice or four times that of the ubiquitous 'double' data type. When multiple representations are available, the compiler can select, e.g., the smallest representation needed to deliver the desired number of trusted significant digits.

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2544-8/14/01. . . $15.00.
http://dx.doi.org/10.1145/2535838.2535874

To summarize, viewing source code as operating on real numbers has many advantages:

- Programmers can reason about correctness using real arithmetic instead of finite-precision arithmetic. We achieve separation of the design of algorithms (which may still be approximate for other reasons) from their realization using finite-precision computations.
- We can verify the ideal meaning of programs using techniques developed to reason over real numbers, which are more scalable and better understood than techniques that directly deal with finite-precision arithmetic.
- The approach allows us to quantify the deviation of implementation outputs from ideal ones, instead of merely proving e.g. range bounds of floating-point variables which is used in simpler static analyses.
- The compiler for reals is free to do optimizations as long as they preserve the precision requirements. This allows the compiler to apply, for example, associativity of arithmetic [17], or even select different approximation schemes for transcendental functions.
- In addition to roundoff errors, the approach also allows the developer to quantify program behavior in the face of external uncertainties such as input measurement errors.

Using our verification conditions, the correctness and the precision of compilation for small programs can, in principle, be directly verified using an SMT solver such as Z3 [20] (see Section 5.3). The capabilities of such solvers are likely continue to improve as the solvers advance, benefiting our approach. However, the complexity of the generated verification conditions for larger programs is currently out of reach of such solvers and we believe that specialized techniques are and will continue to be necessary for this task. This paper presents two specialized techniques that improve the feasibility of the verification task. The first technique performs local approximation and is effective even in benchmarks containing nonlinear arithmetic. The second technique specifically handles conditional expressions.

1.1 Solving Non-Linear Constraints. Forward Propagation

Nonlinear arithmetic poses a significant challenge for verification because it cannot directly be handled using Simplex-like algorithms embedded inside SMT solvers. Although interesting relevant fragments are decidable and are supported by modern solvers, the complexity of solving such constraints is much higher, in terms of both worst-case complexity and the experience in practice. Unfortunately, non-linear arithmetic is ubiquitous in numerical software. Furthermore, our verification conditions add roundoff error terms to arithmetic expressions, so the resulting constraints grow further in complexity, often becoming out of reach of solvers. An alternative to encoding into SMT solver input is to use a sound and over-approximating arithmetic model such as interval or affine arithmetic [19]. However, when used by itself on non-linear code, these approaches yield too pessimistic results, failing to establish any bounds on precision in a number of useful benchmarks.

We show that we can combine range arithmetic computation with SMT solving to overcome the limitations of each of the individual techniques. From the point of view of the logical encoding of the problem, range arithmetic becomes a specialized method to perform approximate quantifier elimination of bounded variables that describe the uncertainties. We obtain a sound, precise, and somewhat scalable procedure. During range computation, our technique also checks for common problems such as overflow, division by zero or square root of a negative number, emitting the corresponding warnings. Because the procedure is a forward computation, it is suitable for automatically generating function summaries containing output ranges and errors of a function. This is a feature that SMT solvers do not solve by themselves, because their primary functionality is answering formula satisfiability questions.

1.2 Sound Compilation of Conditionals

In the presence of uncertainties, conditional branches become another verification challenge. Namely, the ideal execution may follow one branch, but, because of input or roundoff errors, the actual execution follows another. This behavior may be acceptable, however, if we can show that the error on the output remains within required bounds. Our approach could benefit from modular automated analysis of continuity, which was advocated previously [53]. Because we are interested in concrete bounds, we present a new method to check that different paths taken by real-valued and finite-precision versions of the program still preserve the desired precision specification. Our check does not require continuity (which can be difficult to prove for non-linear code). Instead, it directly checks that the difference between the two values on different branches meets the required precision. This technique extends our method for handling non-linear arithmetic, so it benefits from the combination of range arithmetic and SMT solving.

1.3 Implementation and Evaluation

We have implemented our compilation and verification procedure, including the verification condition generation, analysis of possibly non-linear expressions, and the handling of conditionals. Our system is implemented as an extension of the Leon verifier for functional Scala programs [7]. The implementation relies on a range arithmetic implementation [15] for Scala as well as on the Z3 SMT solver [20]. We have evaluated the system on a number of diverse benchmarks, obtaining promising results. Our implementation and the benchmarks are available from

`http://lara.epfl.ch/w/rosa`

To support programming of larger code fragments, our system also supports a basic modular verification technique, which handles functions by replacing calls with function postconditions or by inlining bodies of called functions. We thus expect that our technique is applicable to larger code bases as well, possibly through refactoring code into multiple smaller and annotated functions. Even on the benchmarks that we release, we are aware of no other available system that would provide the same guarantees with our level of automation.

1.4 Summary of Contributions

Our overall contribution is an approach for sound compilation of real numbers into finite-precision representation. Specifically:

- We present a real-valued implementation and specification language for numerical programs with uncertainties; we define its semantics in terms of verification constraints that they induce. We believe that such verification conditions can be used as a standardized format for verifying the precision and the correctness of numerical programs.
- We develop an approximation procedure for computing precise range and error bounds for nonlinear expressions which combines SMT solving with interval arithmetic. We show that such an approach significantly improves computed range and error bounds compared to standard interval arithmetic, and scales better than SMT solving alone. Our procedure can also be used independently as a more precise alternative to interval arithmetic, and thus can perform forward computation without requiring function postconditions to be provided.

- We describe an approach for soundly computing error bounds in the presence of branches and uncertainties, which ensures soundness of compilation in case the function defined by a program with conditionals.
- We have implemented our framework and report our experience on a set of diverse benchmarks, including benchmarks from physics, biology, chemistry, and control systems. The results show that our technique is effective and that it achieves a synergy of the techniques on which it relies.

2. Example

We demonstrate some aspects of our system on the example written in the Scala programming language [45] in Figure 1. The methods triangle and triangleSorted compute the area of a triangle with side lengths a, b and c. We consider a particular application where the user may have two side lengths given, and may vary the third. She has two functions available to do the computation and wants to determine whether either or both satisfy the precision requirement of 1e−11 on line 8. **require** and **ensuring** give the pre- and postconditions of functions, which are written in Scala. Notation res +/− 1e−11 denotes that the return value res should have an absolute error of at most 1e−11 compared to the ideal computation over reals. Our tool determines that such requirement needs at least double floating-point precision and emits the corresponding code. In general, the challenge is to establish that this precision is sufficient to ensure the required bounds, given that errors in finite-precision code accumulate and grow without an *a priori* bound.

Our tool verifies fully automatically that the method triangleSorted indeed satisfies the postcondition and generates the source code with the Double data type which also includes a more precise and complete postcondition on main:

```
0.01955760940017617 ≤ res ∧ res ≤ 12.51998402556971 ∧
  res +/− 8.578997409317759e−12
```

To achieve this result, our tool first checks that the precondition of the function call is satisfied using the Z3 solver. Then, it inlines the body of the function triangleSorted and computes a sound bound on the result's uncertainty with our approximation procedure. It uses the computed bounds to show that the postcondition of main is satisfied. The error computation takes into account in a sound way the input uncertainty (here: an initial roundoff error on the inputs), its propagation, and roundoff errors committed at each arithmetic operation. Additionally, due to the roundoff error, the comparison on line 21 may give a different truth value in boundary cases. Certain floating-point computations will therefore take a different branch than their corresponding real-valued computation. More precisely, the total error when computing the condition is $2e-15$, as computed by our tool. That is, floating-point values that satisfy $a < b + 2e - 15$ may take the **else** branch, even though the corresponding real values would follow the then branch, and similarly in the opposite direction. Our tool verifies that, despite this phenomenon, the difference in the computed result in two branches remains within the precision requirement. Intuitively, the values of computed branches are close for the interval where the truth value of the condition changes, and these values in real-valued and finite-precision implementation are close to each other. Finally, our tool uses our novel range computation procedure to also find a more precise output range than we could have obtained in, e.g., interval arithmetic or any static analysis method that does not differentiate roundoff error variations from the fact that a program can be ran over an interval of inputs. Our tool computes the range $[0.0195, 12.52]$ for both methods, but shows a difference in the absolute error of the computation. For the method triangle, the verification fails, because the computed error $(2.3e-11)$ exceeds the required precision bound. This result is expected—the textbook formula for triangles is known to suffer from imprecision for flat triangles [35], which is somewhat rectified in the method triangleSorted. On the other hand, our tool proves that using triangleSorted delivers the desired precision of 10^{-11} on the result.

```
   def main(a: Real, b: Real, c: Real): Real = {
 2   require(4.500005 ≤ a && a ≤ 6.5)
     val b = 4.0
 4   val c = 8.5
     //val area = triangle(a, b, c)
 6   val area = triangleSorted(a, b, c)
     area
 8 } ensuring(res ⇒ res +/− 1e−11)

10 def triangle(a: Real, b: Real, c: Real): Real = {
     require(1 < a && a < 9 && 1 < b && b < 9 && 1 < c && c < 9 &&
12     a + b > c + 1e−6 && a + c > b + 1e−6 && b + c > a + 1e−6)
     val s = (a + b + c)/2.0
14   sqrt(s ∗ (s − a) ∗ (s − b) ∗ (s − c))
   }
16
   def triangleSorted(a: Real, b: Real, c: Real): Real = {
18   require(1 < a && a < 9 && 1 < b && b < 9 && 1 < c && c < 9 &&
       a + b > c + 1e−6 && a + c > b + 1e−6 && b + c > a + 1e−6 &&
20     a < c && b < c)
     if (a < b) {
22     sqrt((c+(b+a)) ∗ (a−(c−b)) ∗ (a+(c−b)) ∗ (c+(b−a)))/4.0
     } else {
24     sqrt((c+(a+b)) ∗ (b−(c−a)) ∗ (b+(c−a)) ∗ (c+(a−b))) / 4.0
     }
26 }
```

Figure 1. Computing the area of a triangle with a given precision.

3. Programs with Reals

Each program to be compiled consists of one top-level object with methods written in a functional subset of the Scala programming language [45]. All methods are functions over the **Real** data type and the user annotates them with pre- and postconditions that explicitly talk about uncertainties. **Real** represents ideal real numbers without any uncertainty. We allow arithmetic expressions over **Real**s with the standard arithmetic operators $\{+, -, *, /, \sqrt{}\}$, and together with conditionals and function calls they form the body of methods. Our tool also supports immutable variable declarations as **val** x = ... This language allows the user to define a computation over real numbers. Note that this specification language is not executable.

The precondition allows the user to provide a specification of the environment. A complete environment specification consists of lower and upper bounds for all method parameters and an upper bound on the uncertainty or noise. Range bounds are expressed with regular comparison operators. Uncertainty is expressed with a predicate such as x +/− 1e−6, which denotes that the variable x is only known up to $1e-6$. Alternatively, the programmer can specify the relative error as x +/− 1e−7 ∗ x. If no noise except for roundoff is present, roundoff errors are automatically added to input variables.

The postcondition can specify constraints on the output, and in particular the range and the maximum accepted uncertainty. In addition to the language allowed in the precondition, the postcondition may reference the errors on inputs directly in the following way: res +/− 3.5 ∗ !x, which says that the maximum acceptable error on the output variable res is bounded from above by 3.5 times the initial error on x. Whereas the precondition may only talk about the ideal values, the postcondition can also reference the actual value directly via ∼x. This allows us to assert that runtime values will not exceed a certain range, for instance.

Floating-point arithmetic Our tool and technique support in the generated target code any floating-point precision and in particular, single and double floating-point precision as defined by the IEEE 754 floating-point standard [51]. We assume rounding-to-nearest rounding mode and that basic arithmetic operations $\{+, -, *, /, \sqrt{}\}$ are rounded correctly, which means that the result from any such operation must be the closest representable floating-point number. Hence, provided there is no overflow and the numbers are not in denormal range, the result of a binary operation $\circ_F$ satisfies

$$x \circ_F y = (x \circ_R y)(1 + \delta), \quad |\delta| \leq \epsilon_M, \quad \circ \in \{+, -, *, /\} \qquad (1)$$

where $\circ_R$ is the ideal operation in real numbers and ϵ_M is the machine epsilon that determines the upper bound on the relative error. This model provides a basis for our roundoff error estimates. When there is a possibility of an overflow or denormal values, our analysis reports an error. An extension of the analysis to handle denormals is straightforward, by adding an additional error term in Equation 1.

Fixed-point arithmetic Our tool and technique also support standard fixed-point arithmetic; for more details see [3]. Our precision analysis supports any bit-width. Our code generator generates code for 16 and 32 bit fixed-point arithmetic, which are the most common choices, using integers and bit-shifts.

Other finite-precision representations of rationals The techniques described in this paper are general in that they are also applicable to other arithmetic representations, as long as roundoff errors can be computed at each computation step from the ranges of variables. Examples include floating-point implementations with a different number of bits for the exponent and mantissa, or redundant arithmetic.

4. Compiling Reals to Finite Precision

Given a specification or program over reals and possible target data type(s), our tool generates code over floating-point or fixed-point numbers that satisfy the given pre- and postconditions (and thus meet the target precision). Figure 2 presents a high-level view of our compilation algorithm. Our tool first analyses the entire specification and generates one verification condition for each postcondition to be proven. To obtain a modular algorithm, the tool also generates verification conditions that check that at each function call the precondition of the called function is satisfied. The methods are then sorted by occurring function calls. This allows us to reuse already computed postconditions of function calls in a modular analysis. If the user specifies one target data type, the remaining part of the compilation process is performed with respect to this data type's precision. If not or in the case the user specified several possible types, our tool will perform a binary search over the possible types to find the least in the list that satisfies all specifications. The user of our tool can provide the list of possible data types manually and sort them by her individual preference. Currently, the analysis is performed separately for each data type, which is not a big issue due to the relatively small number of alternatives. We did identify certain shared computations between iterations; we can exploit them in the future for more efficient compilation. In order for the compilation process to succeed, the specification has to be met with respect to a given finite-precision arithmetic, thus the principal part of our algorithm is spent in verification, which we describe in Section 5.

We envision that in the future the compilation task will also include automatic precision-preserving code optimizations, but in this paper we concentrate on the challenging groundwork of verifying the precision of code.

```
Input: spec: specification over Reals, prec: candidate precisions
for fnc ← spec.fncs
  fnc.vcs = generateVCs(fnc)

spec.fncs.sortBy((f1, f2) => f1 ⊆ f2.fncCalls)

while prec ≠ ∅ and notProven(spec.fncs)
  precision = prec.nextPrecise
  for fnc ← spec.fncs
    for vc ← fnc.vcs
      while vc.hasNextApproximation ∧ notProven(vc)
        approx = getNextApproximation(vc, precision)
        vc.status = checkWithZ3(approx)
  generateSpec(fnc)
generateCode(spec)

Output: floating−point or fixed−point code
```

Figure 2. Compilation algorithm.

Our tool can currently generate Scala code over

- fixed-point arithmetic with a 16 or 32 bit width, or
- floating-point arithmetic in single (32 bit), double (64 bit), double-double (128 bit) and quad-double (256 bit) precision.

We currently do not support mixing of data types in one program, but plan to explore this avenue in the future. For double-double and quad-double precision, which were implemented in software by [5], we provide a Scala interface to the library with the generated code. In case the verification part of compilation fails, our tool prints a failure report with the best postconditions our tool was able to compute. The user can then use the generated specifications to gain insight why and where her program does not satisfy the requirements.

While we have implemented our tool to accept specifications in a domain specific language embedded in Scala and generate code in Scala, all our techniques apply equally to all programming languages and hardware that follow the floating-point abstraction we assume (Equation 1).

5. Verifying Real Programs

We will now describe the verification part of our compilation algorithm. In the following we will call the *ideal* computation the computation in the absence of any uncertainties and implemented in a real arithmetic, and the *actual* computation the computation that will finally be executed in finite-precision and with potentially uncertain inputs.

5.1 Verification Conditions for Loop-Free Programs

For each method with a precondition P and a postcondition Q our approach considers the following verification condition:

$$\forall \vec{x}, \vec{res}, \vec{y}.\ P(\vec{x}) \wedge body(\vec{x}, \vec{y}, \vec{res}) \rightarrow Q(\vec{x}, \vec{res}) \qquad (*)$$

where $\vec{x}, \vec{res}, \vec{y}$ denote the input, output and local variables respectively. Table 1 summarizes how verification constraints are generated from our specification language for floating-point arithmetic. Each variable x in the specification corresponds to two real-valued variables $x, x_\circ$, the ideal one in the absence of uncertainties and roundoff errors and the actual one, computed by the compiled program. Note that the ideal and actual variables are related only through the error bounds in the pre- and postconditions, which al-

a <= x && x <= b	$x \in [a, b]$
x +/− k	$x_\circ = x + err_x \wedge err_x \in [-k, k]$
x +/− m ∗ x	$x_\circ = x + err_x \wedge err_x \in [-\|mx\|, \|mx\|]$
∼x	$x_\circ$
!x	err_x
x ⋄ y (ideal part)	$(x \diamond y)$
x ⋄ y (actual part)	$(x_\circ \diamond y_\circ)(1 + \delta_1)$
sqrt(x) (ideal part)	$sqrt(x)$
sqrt(x) (actual part)	$sqrt(x_\circ)(1 + \delta_2)$
val z = x	$z = x \wedge z_\circ = x_\circ$
if (c(x)) e1(x)	$((c(x) \wedge e_1(x)) \vee (\neg c(x) \wedge e_2(x)))\wedge$
else e2(x)	$((c_\circ(x_\circ) \wedge e_{1\circ}(x_\circ)) \vee (\neg c_\circ(x_\circ) \wedge e_{2\circ}(x_\circ)))$
g(x)	$g(x) \wedge g_\circ(x_\circ)$

$\diamond \in \{+, -, *, /\}$

$-\epsilon_m \leq \delta_i \wedge \delta_i \leq \epsilon_m$, all δ are fresh

$cond_\circ$ and $e_\circ$ denote functions with roundoff errors at each step

Table 1. Semantics of our specification language.

lows for the ideal and actual executions to take different paths. In the method body we have to take into account roundoff errors from arithmetic operations and the propagation of existing errors. Our system currently supports operations $\{+, -, *, /, \sqrt{}\}$, but these can be in principle extended to elementary functions, for instance by encoding them via Taylor expansions [41]. Note that the resulting verification conditions are parametric in the machine epsilon.

For fixed-point arithmetic constraints such as above can also be generated, although the translation is more complex due to the fact that fixed-point formats have to be statically determined. Roundoff error can thus no longer be encoded by a formula such as Equation 1. One possible translation is to speculatively assign fixed-point formats (including the position of the fixed point) based on real-valued ranges, and then verify that the resulting constraint with the assigned formats still holds in the presence of roundoff errors. However, because a direct attempt to solve verification conditions using an off-the-shelf solver alone is not satisfactory (see Section 5.3), our tool directly uses the approximation procedure from Section 5.4 and computes sound ranges for fixed-point implementation, allocating fixed-point formats on the fly.

5.2 Specification Generation

In order to give feedback to developers and to facilitate automatic modular analysis, our tool also provides automatic specification generation. By this we mean that the programmer still needs to provide the environment specification in form of preconditions, but our tool automatically computes a precise postcondition.

Formally, we can rewrite the constraint (*) as

$$\forall \vec{x}, res.\ (\exists \vec{y}.\ P(\vec{x}) \wedge body(\vec{x}, \vec{y}, res)) \rightarrow Q(\vec{x}, res)$$

where Q is now unknown. We obtain the most precise postcondition Q by applying quantifier elimination (QE) to $P(\vec{x}) \wedge body(\vec{x}, \vec{y}, res)$ and eliminate $\vec{y}$. The theory of arithmetic over reals admits QE so it is theoretically possible to use this approach.
We do not currently use a full QE procedure for specification generation, as it is expensive and it is not clear whether the returned expressions would be of a suitable format. Instead, we use our approximation approach which computes ranges and maximum errors in a forward fashion and computes an (over) approximation of a postcondition of the form $res \in [a, b] \wedge res \pm u$. When proving a postcondition, our tool automatically generates these specifications and provides them as feedback to the user.

def getNextApproximation(vc, precision):

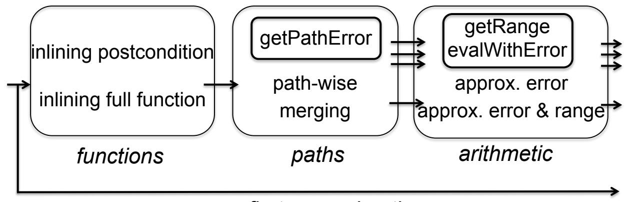

Figure 3. Approximation pipeline.

5.3 Difficulty of Simple Encoding into SMT solvers

For small functions we can already prove interesting properties by using the exact encoding of the problem just described and discharging the verification constraints with Z3. Consider the following code a programmer may write to implement the third B-spline basic function which is commonly used in signal processing [33].

```
def bspline3(u: Real): Real = {
  require(0 ≤ u && u ≤ 1 && u +/− 1e−13)
  −u∗u∗u / 6.0
} ensuring (res ⇒ −0.17 ≤ res && res ≤ 0.05 && res +/− 1e−11)
```

Functions and the corresponding verification conditions of this complexity are already within the possibilities of the nonlinear solver within Z3. For more complex functions however, Z3 does not (yet) provide an answer in a reasonable time, or returns unknown. Whether alternative techniques in SMT solvers can help in such cases remains to be seen [10, 34]. We here provide an approach based on step-wise approximation that addresses the difficulty of general-purpose constraint solving.

5.4 Verification with Approximations

To soundly compile more interesting programs, we have developed an approximation procedure that computes a sound over-approximation of the range of an expression and of the uncertainty on the output. This procedure is a forward computation and we also use it to generate specifications automatically. We describe the approximation procedure in detail in Section 6, for now we will assume that it exists and, given a precondition P and an expression $expr$, computes a sound bound on the output range and its associated uncertainty:

$$([a, b], err) = \text{evalWithError}(P, expr) \Rightarrow$$
$$\forall \vec{x}, \vec{x}_\circ, res, res_\circ. P(\vec{x}, \vec{x_\circ}) \wedge res = expr(\vec{x}) \wedge res_\circ = expr_\circ(\vec{x}_\circ)$$
$$\rightarrow (a \leq res \vee res \leq b) \wedge |res - res_\circ| < err$$

We have identified three possibilities for approximation: nonlinear arithmetic, function calls, and paths due to conditionals and each can be approximated at different levels. We have observed in our experiments, that "one size does not fit all" and a combination of different approximations is most successful in proving the verification conditions we encountered. For each verification condition we thus construct approximations until Z3 is able to prove one, or until we run out of approximations where we report the verification as failed. We can thus view verification as a stream of approximations to be proven. We illustrate the pipeline that computes the different approximations in Figure 3. The routines getPathError, getRange and evalWithError are described in the following sections in more detail.

The first approximation (indicated by the long arrow in Figure 3) is to use Z3 alone on the entire constraint constructed by the rules in Table 1. This is indeed an approximation, as all function calls are treated as uninterpreted functions in this case. As noted before, this approach only works in very simple cases or when no uncertainties and no functions are present. Then, taking all possible combinations of subcomponents in our pipeline we obtain the other approximations, which are filtered accordingly depending on whether the constraint contains function calls or conditional branches.

Function calls If the verification constraint contains function calls and the first approximation failed, our tool will attempt to inline postconditions and pass on the resulting constraint down the approximation pipeline. We support inlining of both user-provided postconditions and postconditions computed by our own specification generation procedure. If this still is not precise enough, we inline the entire function body.

Postcondition inlining is implemented by replacing the function call with a fresh variable and constraining it with the postcondition. Thus, if verification succeeds with inlining the postcondition, we avoid having to consider each path of the inlined function separately and can perform modular verification avoiding a potential path explosion problem. Such modular verification is not feasible when postconditions are too imprecise and we plan to explore the generation of more precise postconditions in the future. One step in this direction is to allow postconditions that are parametric in the initial errors, for example with the operator $!x$ introduced in Section 3. While our tool currently supports postcondition inlining with such postconditions, we do not yet generate these automatically.

Arithmetic The arithmetic part of the verification constraints generated by Table 1 can be essentially divided into the ideal part and the actual part, which includes roundoff errors at each computation step. The ideal part determines whether the ideal range constraints in the postcondition are satisfied and the actual part determines whether the uncertainty part of the postcondition is satisfied. We can use our procedure presented in Section 6 to compute a sound approximation of both the result's range as well as its uncertainty.

Based on this, our tool first constructs an approximation which leaves the ideal part unchanged, but replaces the actual part of the constraint by the computed uncertainty bound. This effectively removes a large number of variables and is many times a sufficient simplification for Z3 to succeed in verifying the entire constraint. If Z3 is still not able to prove the constraint, our tool constructs the next approximation by also replacing the ideal part, this time with a constraint of the result's range which has been computed by our approximation procedure previously. Note that this second approximation may not have enough information to prove a more complex postcondition, as correlation information is lost. We note that the computation of ranges and errors is the same for both approximations and thus trying both does not affect efficiency significantly. In our experiments, Z3 is able to prove the ideal, real-valued part in most cases, so this second approximation is rarely used.

Paths In the case of several paths through the program, we have the option to consider each path separately or to merge results at each join in the control flow graph. This introduces a tradeoff between efficiency and precision, since on one hand, considering each path separately leads to an exponential number of paths to consider. On the other hand, merging at each join looses correlation information between variables which may be necessary to prove certain properties. Our approximation pipeline chooses merging first, before resorting to a path-by-path verification in case of failure. We believe that other techniques for exploring the path space could also be integrated into our tool [12, 36]. Another possible improvement are heuristics that select a different order of approximations depending on particular characteristics of the verification condition.

```
def comparisonValid(x: Real): Real = {
  require(−2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 − z2
} ensuring(res ⇒ res <= 0.1 && res +/− 5e−14)

def comparisonInvalid(x: Real): Real = {
  require(−2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 − z2
} ensuring(res ⇒ res <= 0.01 && res +/− 5e−14)

def sineTaylor(x: Real): Real = {
  require(−2.0 < x && x < 2.0)
  x − (x∗x∗x)/6.0 + (x∗x∗x∗x∗x)/120.0 − (x∗x∗x∗x∗x∗x∗x)/5040.0
} ensuring(res => −1.0 < res && res < 1.0 && res +/− 1e−14)

def sineOrder3(x: Real): Real = {
  require(−2.0 < x && x < 2.0)
  0.954929658551372 ∗ x − 0.12900613773279798∗(x∗x∗x)
} ensuring(res ⇒ −1.0 < res && res < 1.0 && res +/− 1e−14)
```

Figure 4. Different polynomial approximations of sine.

Example We illustrate the verification algorithm on the example in Figure 4, using double floating-point precision as the target. The functions sineTaylor and sineOrder3 are verified first since they do not contain function calls. Verification with the full verification constraint fails. Next, our tool computes the errors on the output and Z3 succeeds to prove the resulting constraint with the ideal part untouched. From this approximation our tool directly computes a new, more precise postcondition, in particular it can narrow the resulting errors to `1.63e-15` and `1.11e-15` respectively. Next, our tool considers the comparisonValid function. Inlining only the postcondition is not enough in this case, but computing the error approximation on the inlined functions succeeds in verifying the postcondition. Note that our tool does not approximate the real-valued portion of the constraint, i.e. Z3 is used directly to verify the constraint $z1 - z2 \leq 0.1$. This illustrates our separation of the real reasoning from the finite-precision implementation: with our separation we can use a real arithmetic solver to deal with algorithmic reasoning and verify with our error computation that the results are still valid (within the error bounds) in the implementation. Finally, the tool verifies that the preconditions of the function calls are satisfied by using Z3 alone. Verification of the function comparisonInvalid fails with all approximations. Our tool is able to determine that the ideal real-valued constraint alone ($z1 - z2 \leq 0.01$) is not valid, reports a counterexample ($x = 1.875$) and returns invalid as the verification result.

5.5 Soundness

Our procedure is sound because our constraints over-approximate the actual errors. Furthermore, even in the full constraint as generated from Table 1, roundoff errors are over-approximated since we assume the worst-case error bound at each step. While this ensures soundness, it also introduces incompleteness, as we may fail to validate a specification because our over-approximation is too large. This implies that counterexamples reported by Z3 are in general only valid, if they disprove the ideal real-valued part of the verification constraint. Our tool checks whether this is the case by constructing a constraint with only the real-valued part, and reports the counterexamples, if such are returned from Z3.

5.6 Loops and Recursion

In principle, our techniques can be applied to programs with loops via recursion, however, because of accumulation of roundoff errors only self-stabilizing systems can be expected to have simple inductive invariants, and such systems can to some extent also be addressed using runtime verification techniques [16]. For non-self-stabilizing systems the uncertainties depend on the number of iterations, which makes specifications of such functions very complex. Note that, for ensuring the stability of certain embedded control systems, it has been shown that it is sufficient to consider the body of the control loop only [3].

6. Solving Nonlinear Constraints

Having given an overview of the approximation pipeline, we now describe the computation of the approximation for nonlinear arithmetic, which corresponds to the last box in Figure 3. For completeness of presentation, we first review interval and affine arithmetic which are common choices for performing sound arithmetic computations and which we also use as part of our technique. We then present our novel procedure for computing the output range of a nonlinear expression given ranges for its inputs that can be a more precise substitute for interval or affine arithmetic. Finally, we continue with a procedure that computes a sound over-approximation of the uncertainty on the result of a nonlinear expression.

One possibility to perform guaranteed computations is to use standard interval arithmetic [43]. Interval arithmetic computes a bounding interval for each basic operation as

$$x \circ y = [min(x \circ y), max(x \circ y)] \qquad \circ \in \{+, -, *, /\}$$

and analogously for square root.

Affine arithmetic was originally introduced in [19] and addresses the difficulty of interval arithmetic in handling correlations between variables. Affine arithmetic represents possible values of variables as affine forms

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i$$

where x_0 denotes the *central value* (of the represented interval) and each *noise symbol* ϵ_i is a formal variable denoting a deviation from the central value, intended to range over $[-1, 1]$. The maximum magnitude of each *noise term* is given by the corresponding x_i. Note that the sign of x_i does not matter in isolation, it does, however, reflect the relative dependence between values. E.g., take $x = x_0 + x_1\epsilon_1$, then

$$\begin{aligned} x - x &= x_0 + x_1\epsilon_1 - (x_0 + x_1\epsilon_1) \\ &= x_0 - x_0 + x_1\epsilon_1 - x_1\epsilon_1 = 0 \end{aligned}$$

If we subtracted $x' = x_0 - x_1\epsilon_1$ instead, the resulting interval would have width $2 * x_1$ and not zero.

The range represented by an affine form is computed as

$$[\hat{x}] = [x_0 - rad(\hat{x}), x_0 + rad(\hat{x})], \qquad rad(\hat{x}) = \sum_{i=1}^{n} |x_i|$$

A general affine operation $\alpha\hat{x} + \beta\hat{y} + \zeta$ consists of addition, subtraction, addition of a constant (ζ) or multiplication by a constant (α, β). Expanding the affine forms $\hat{x}$ and $\hat{y}$ we get

$$\alpha\hat{x} + \beta\hat{y} + \zeta = (\alpha x_0 + \beta y_0 + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i$$

```
def getRange(expr, precondition, precision, maxIterations):
  z3.assertConstraint(precondition)
  [aInit, bInit] = evalInterval(expr, precondition.ranges);

  //lower bound
  if z3.checkSat(expr < a + precision) == UNSAT
    a = aInit
    b = bInit
    numIterations = 0
    while (b−a) < precision ∧ numIterations < maxIterations
      mid = a + (b − a) / 2
      numIterations++
      z3.checkSat(expr < mid) match
        case SAT ⇒ b = mid
        case UNSAT ⇒ a = mid
        case Unknown ⇒ break
    aNew = a
  else
    aNew = aInit

  //upper bound symmetrically
  bNew = ...
  return: [aNew, bNew]
```

Figure 5. Algorithm for computing the range of an expression.

An additional motivation for using affine arithmetic is that different contributions to the range it represents remain, at least partly, separated. This information can be used for instance to help identify the major contributor of a result's uncertainty or to separate contributions from external uncertainties from roundoff errors.

6.1 Range Computation

The goal of this procedure is to perform a forward-computation to determine the real-valued range of a nonlinear arithmetic expression given ranges for its inputs. Two common possibilities are interval and affine arithmetic, but they tend to over-approximate the resulting range, especially if the input intervals are not sufficiently small (order 1). Affine arithmetic improves over interval arithmetic somewhat by tracking linear correlations, but in the case of nonlinear expressions the results can become actually worse than for interval arithmetic (e.g. $x * y$, where $x = [-5, 3], y = [-3, 1]$ gives $[-13, 15]$ in affine arithmetic and $[-9, 15]$ in interval arithmetic).

Observation: A nonlinear theorem prover such as the one that comes with Z3 can decide with fairly good precision whether a given bound is sound or not. That is we can check with a prover whether for an expression e the range $[a, b]$ is a sound interval enclosure. This observation is the basis of our range computation.

The input to our algorithm is a nonlinear expression *expr* and a precondition P on its inputs, which specifies, among possibly other constraints, ranges on all input variables $\vec{x}$. The output is an interval $[a, b]$ which satisfies the following:

$$\begin{aligned} &[a, b] = \text{getRange}(P, expr) \Rightarrow \\ &\qquad \forall \vec{x}, res. P(\vec{x}) \wedge res = expr(\vec{x}) \rightarrow (a \le res \vee res \le b) \end{aligned}$$

The algorithm for computing the lower bound of a range is given in Figure 5. The computation for the upper bound is symmetric. For each range to be computed, our tool first computes an initial sound estimate of the range with interval arithmetic. It then performs an initial quick check to test whether the computed first approximation bounds are already tight. If not, it uses the first approximation as the starting point and then narrows down the lower and upper bounds using a binary search. At each step of the binary search our tool uses Z3 to confirm or reject the newly proposed bound.

The search stops when either Z3 fails, i.e. returns unknown for a query or cannot answer within a given timeout, the difference between subsequent bounds is smaller than a precision threshold, or the maximum number of iterations is reached. This stopping criterion can be set dynamically.

Additional constraints In addition to the input ranges, the precondition may also contain further constraints on the variables. For example consider again the method triangle in Figure 1. The precondition bounds the inputs as $a, b, c \in [1, 9]$, but the formula is useful only for valid triangles, i.e. when every two sides together are longer than the third. If not, we will get an error at the latest when we try to take the square root of a negative number. In interval-based approaches we can only consider input intervals that satisfy this constraint for all values, and thus have to check several (and possibly many) cases. In our approach, since we are using Z3 to check the soundness of bounds, we can assert the additional constraints up-front and then all subsequent checks are performed with respect to all additional and initial constraints. This allows us to avoid interval subdivisions due to imprecisions or problem specific constraints such as those in the triangle example. This becomes especially valuable in the presence of multiple variables, where we may otherwise need an exponential number of subdivisions.

6.2 Error Approximation

We now describe our approximation procedure which, for a given expression $expr$ and a precondition P on the inputs, computes the range and error on the output. More formally, our procedure satisfies the following:

$$([a, b], err) = \text{evalWithError}(P, expr) \Rightarrow$$
$$\forall \vec{x}, \vec{x_\circ}, res, res_\circ . P(\vec{x}, \vec{x_\circ}) \wedge res = expr(\vec{x}) \wedge res_\circ = expr_\circ(\vec{x_\circ})$$
$$\rightarrow (a \leq res \vee res \leq b) \wedge |res - res_\circ| < err$$

where $expr_\circ$ represents the expression evaluated in finite-precision arithmetic and $\vec{x}, \vec{x_\circ}$ are the ideal and actual variables. The precondition specifies the ranges and uncertainties of initial variables and other additional constraints on the ideal variables. The uncertainty specification is necessary, as it relates the ideal and actual variables.

The idea of our procedure is to "execute" a computation while keeping track of the output range of the current expression and its associated errors. At each arithmetic operation, we propagate existing errors, compute an upper bound on the roundoff error and add it to the overall errors. Since the roundoff error depends proportionally on the range of values, we need to keep track of the ranges as precisely as possible.

Our procedure is build on the abstraction that a computation is an ideal computation plus or minus some uncertainty. The abstraction of floating-point roundoff errors that we choose also follows this separation:

$$fl(x \diamond y) = (x \diamond y)(1 + \delta) = (x \diamond y) + (x \diamond y)\delta$$

for $\delta \in [-\epsilon_m, \epsilon_m]$ and $\diamond \in \{+, -, *, /\}$. This allows us to treat all uncertainties in a unified manner. For fixed-point arithmetic the situation is similar, but we first determine the fixed-point format from the current range, and only from this compute the roundoff error.

Our procedure builds on the idea of the SmartFloat data type [15], which uses affine arithmetic to track both the range and the errors. For nonlinear operations, however, the so computed ranges become quickly very pessimistic and the error computation may also suffer from this imprecision. We observed that since the errors tend to be relatively small, this imprecision does not affect the error propagation itself to such an extent. If the initial errors are small (less than one), multiplied nonlinear terms tend to be even smaller, whereas if the affine terms are larger than one, the nonlinear terms grow. We thus concentrate on improving the ideal range of values and use our novel range computation procedure for this part and leave the error propagation with affine arithmetic as in [15].

In our adaptation, we represent every variable and intermediate computation result as a datatype with the following components:

$$x : (range : Interval, \hat{err} : AffineForm)$$

where $range$ is the range of this variable, computed as described in Section 6.1 and $\hat{err}$ is the affine form representing the errors. The (over-approximation) of the actual range including all uncertainties is then given by $totalRange = range + [\hat{err}]$, where $\hat{err}$ denotes the interval represented by the affine form.

Roundoff error computation Roundoff errors for floating-point arithmetic are computed at each computation step as

$$\rho = \delta * maxAbs(totalRange)$$

where δ is the machine epsilon, and added to $\hat{err}$ as a fresh noise term. Note that this roundoff error computation makes our error computation parametric in the floating-point precision. For fixed-point arithmetic, roundoff errors are computed as

$$\rho = getFormat(totalRange, bitWidth).quantizationError$$

where the $getFormat$ function returns the best fixed-point format [3] that can accommodate the range. This computation is also parametric in the bit-width.

Error propagation For affine operations addition, subtraction, and multiplication by a constant factor the propagated errors are computed term-wise and thus as for standard affine arithmetic. We refer the reader to [15, 19] for further details and describe here only the propagation for nonlinear arithmetic. For multiplication, division and square root, the magnitude of errors also depends on the ranges of variables. Since our ranges are not affine terms themselves, propagation has to be adjusted. In the following, we denote the range of a variable x by $[x]$ and its associated error by the affine form $\hat{err}_x$. When we write $[x] * \hat{err}_y$ we mean that the interval $[x]$ is converted into an affine form and the multiplication is performed in affine arithmetic.

Multiplication is computed as

$$\begin{aligned} x * y &= ([x] + \hat{err}_x)([y] + \hat{err}_y) \\ &= [x] * [y] + [x] * \hat{err}_y + [y] * \hat{err}_x + \hat{err}_x * \hat{err}_y + \rho \end{aligned}$$

where ρ is the new roundoff error. Thus the first term contributes to the ideal range and the remaining three to the error affine form. The larger the factors $[x]$ and $[y]$ are, the larger the finally computed errors will be. In order to keep the over-approximation as small as possible, we evaluate $[x]$ and $[y]$ with our new range computation.

Division is computed as

$$\begin{aligned} \frac{x}{y} &= x * \frac{1}{y} = ([x] + \hat{err}_x)([1/y] + \hat{err}_{1/y}) \\ &= [x] * [\frac{1}{y}] + [x] * \hat{err}_{\frac{1}{y}} + [\frac{1}{y}] * \hat{err}_x + \hat{err}_x * \hat{err}_{\frac{1}{y}} + \rho \end{aligned}$$

For square root, we first compute an affine approximation of square root as in [15]: $\sqrt{x} = \alpha * x + \zeta + \theta$, and then perform the affine multiplication term wise.

Overflows and NaN Our procedure allows us to detect potential overflows, division by zero and square root of a negative value, as our tool computes ranges of all intermediate values. We currently report these issues as warnings to the user.

6.3 Limitations

The limitation of this approach is clearly the ability of Z3 to check our constraints. We found its capabilities satisfactory, although we expect the performance to still significantly improve. To emphasize the difference to the constraints that are defined by Table 1, the constraints we use here do not add errors at each step and thus the number of variables is reduced significantly. We also found several transformations helpful, such as rewriting powers (e.g. $x * x * x$ to x^3), multiplying out products and avoiding non-strict comparisons in the precondition, although the benefits were not entirely consistent. Note that at each step of our error computation, our tool computes the current range. Thus, even if Z3 fails to tighten the bound for some expressions, we still compute more precise bounds than interval arithmetic overall in most cases, as the ranges of the remaining subexpressions have already been computed more precisely.

7. Conditional Statements

In this Section we consider the difference between the ideal and actual computation due to uncertainties on computing branch conditions and the resulting different paths taken. We note that the full constraint constructed according to Section 3 automatically includes this error. Recall that the ideal and actual computations are independent except for the initial conditions, so that it is possible that they follow different paths through the program.

In the case of using approximation, however, we compute the error on individual paths and have to consider the error due to diverging paths separately. We propose the following algorithm to explicitly compute the difference between the ideal and the actual computation across paths. Note that we do not assume continuity, i.e. the algorithm allows us to compute error bounds even in the case of non-continuous functions.

For simplicity, we present here the algorithm for the case of one conditional statement: **if** (c(x)<0) f1(x) **else** f2(x).
It generalizes readily to more complex expressions. W.l.o.g. we assume that the condition is of the form $c(x) < 0$. Indeed, any conditional of the form $c(x) == 0$ would yield different results for the ideal and actual computation for nearly any input, so we do not allow it in our specification language.

The actual computation commits a certain error when computing the condition of the branch and it is this error that causes some executions to follow a different branch than the corresponding ideal one would take. Consider the case where the ideal computation evaluates f1, but the actual one evaluates f2. The algorithm in Figure 6 gives the computation of the *path error* in this case. The idea is to compute the ranges of f1 and f2, but only for the inputs that could be diverging. The final error is then the maximum difference of these value. The algorithm extends naturally to several variables. In the case of several paths through the program, this error has to be, in principle, computed for each pair of paths. We use Z3 to rule out infeasible paths up front so that the path error computation is only performed for those paths that are actually feasible.

Our tool implements a refined version of this approach, which merges paths to avoid having to consider an exponential number of path combinations. It also uses a higher default precision and number of iterations threshold during the binary search in the range computation as this computation requires in general very tight intervals for each path.

We identify two challenges for performing this computation:

1. As soon as the program has multiple variables, the inputs for the different branches are not two-dimensional intervals anymore, which makes an accurate evaluation of the individual paths difficult.

```
   def getPathError:
2  Input: pre: ($x \in [a, b] \wedge x \pm n$)
          program: (if (cond(x) < 0) f1(x) else f2(x))
4
     val pathError1 = computePathError(pre, cond, f1, f2)
6    val pathError2 = computePathError(pre, ¬ cond, f2, f1)
     return max (pathError1, pathError2)
8
   def computePathError(pre, c, f1, f2):
10   ([c], $err_c$) = evalWithError(pre, c)

12   ([f2]$_{float}$, $err_{float}$) =
       evalWithError(pre ∧ c(x) ∈ [0, $err_c$], f2)
14   [f1]$_{real}$ =
       getRange(pre ∧ c(x) ∈ [$-err_c$, 0], f1)
16   return: max |[f1]$_{real}$ − ([f2]$_{float}$ + $err_{float}$)|
```

Figure 6. Computing error due to diverging paths. Quantities denotes by [x] are intervals.

2. The inputs for the two branches are inter-dependent. Thus, simply evaluating the two branches with inputs that are in the correct ranges, but are not correlated, yields pessimistic results when computing the final difference (line 16).

We overcome the first challenge with our range computation which takes into account additional constraints. For the second challenge, we use our range computation as well. Unfortunately, Z3 fails to tighten the final range to a satisfactory precision due to timeouts. We still obtain much better error estimates than with interval arithmetic alone, as the ranges of values for the individual paths are already computed much more precisely. We report in Section 8 on the type of programs whose verification is within our reach today.

8. Experiments

The examples in Figure 1 and 4 and Section 5.3 provide an idea of the type of programs our tool is currently able to verify fully automatically. The B-spline example from Section 5.3 is the largest meaningful example we were able to find that Z3 alone could verify in the presence of uncertainties. For all other cases, it was necessary to use our approximation methods.

We have chosen several nonlinear expressions commonly used in physics, biology and chemistry [44, 48, 54] as benchmark functions, as well as benchmarks used in control systems [2] and suitable benchmarks from [22]. Experiments were performed on a desktop computer running Ubuntu 12.04.1 with a 3.5GHz i7 processor and 16GB of RAM.

8.1 Compiling Programs

Which data type is suitable for a given program depends on the parameter ranges, the code itself but also the precision required by the application using the code. For example, take the functions in Figure 7. Depending on which precision on the output the user needs, our tool will select different data types. For the requirement res +/− 1e−12, as specified, Double is be a suitable choice for doppler and turbine, however for the jetEngine example this is not sufficient, and thus DoubleDouble would be selected by our tool. The user can influence which data types are preferred by supplying a list to our tool which is ordered by her preference.

Figure 8 illustrates the tradeoff between the precision achieved by different data types against the runtime of the compiled code generated by our tool. We used the Caliper [1] framework for benchmarking the running times.

Benchmark	Our range	interval arithmetic	Simulated range
doppler1	[-137.639, -0.033951]	[-158.720, -0.029442]	[-136.346, -0.035273]
doppler2	[-230.991, -0.022729]	[-276.077, -0.019017]	[-227.841,-0.023235]
doppler3	[-83.066, -0.50744]	[-96.295, -0.43773]	[-82.624, -0.51570]
rigidBody1	[-705.0, 705.0]	[-705.0, 705.0]	[-697.132, 694.508]
rigidBody2	[-56010.1, 58740.0]	[-58740.0, 58740.0]	[-54997.635, 57938.052]
jetEngine	[-1997.037, 5109.338]	$[-\infty, \infty]$	[-1779.551, 4813.564]
turbine1	[-18.526, -1.9916]	[-58.330, -1.5505]	[-18.284, -1.9946]
turbine2	[-28.555, 3.8223]	[-29.437, 80.993]	[-28.528, 3.8107]
turbine3	[0.57172, 11.428]	[0.46610, 40.376]	[0.61170, 11.380]
verhulst	[0.31489, 1.1009]	[0.31489, 1.1009]	[0.36685,0.94492]
predatorPrey	[0.039677, 0.33550]	[0.037277, 0.35711]	[0.039669,0.33558]
carbonGas	[4.3032 e6, 1.6740 e7]	[2.0974 e6, 3.4344 e7]	[4.1508 e6, 1.69074 e7]
Sine	[-0.9999, 0.9999]	[-2.3012, 2.3012]	[-1.0093, 1.0093]
Sqrt	[1.0, 1.3985]	[0.83593, 1.5625]	[1.0, 1.3985]
Sine (order 3 approx.)	[-1.0001, 1.0001]	[-2.9420, 2.9420]	[-1.0, 1.0]

Table 2. Comparison of ranges computed with out procedure against interval arithmetic and simulation. Simulations were performed with 10^7 random inputs. Ranges are rounded outwards. Affine arithmetic does not provide better results than interval arithmetic.

```
def doppler(u: Real, v: Real, T: Real): Real = {
  require(−100 < u && u < 100 && 20 < v && v < 20000 &&
          −30 < T && T < 50)

  val t1 = 331.4 + 0.6 ∗ T
  (− (t1) ∗v) / ((t1 + u)∗(t1 + u))
} ensuring(res ⇒ res +/− 1e−12)

def jetEngine(x1: Real, x2: Real): Real = {
  require(−5 < x1 && x1 < 5 && −20 < x2 && x2 < 5)

  val t = (3∗x1∗x1 + 2∗x2 − x1)
  x1 + ((2∗x1∗(t/(x1∗x1 + 1))∗
  (t/(x1∗x1 + 1) − 3) + x1∗x1∗(4∗(t/(x1∗x1 + 1))−6))∗
  (x1∗x1 + 1) + 3∗x1∗x1∗(t/(x1∗x1 + 1)) + x1∗x1∗x1 + x1 +
  3∗((3∗x1∗x1 + 2∗x2 −x1)/(x1∗x1 + 1)))
} ensuring(res ⇒ res +/− 1e−12)

def turbine(v: Real, w: Real, r: Real): Real = {
  require(−4.5 < v && v < −0.3 && 0.4 < w && w < 0.9 &&
          3.8 < r && r < 7.8)

  3 + 2/(r∗r) − 0.125∗(3−2∗v)∗(w∗w∗r∗r)/(1−v) − 4.5
} ensuring(res ⇒ res +/− 1e−12)
```

Figure 7. Benchmark functions from physics and control systems.

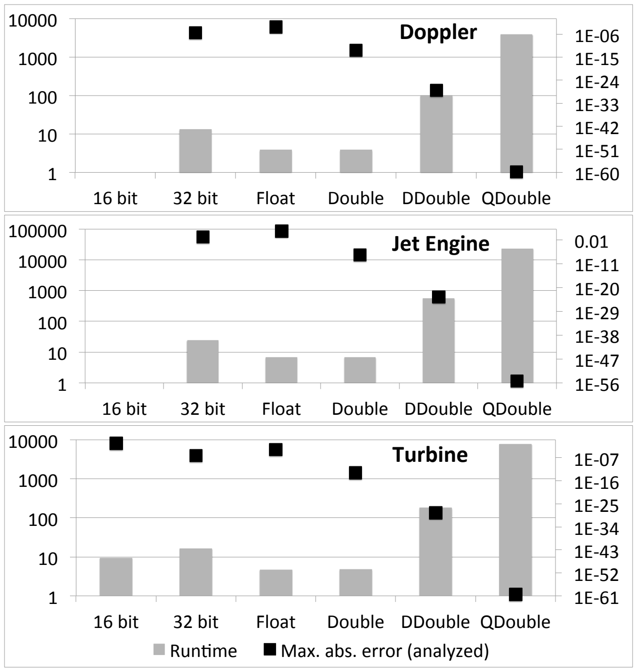

Figure 8. Runtimes of benchmarks compiled for different precisions in nanoseconds (left) vs. abs. error bounds computed for that precision by our tool (right).

For our benchmarks with their limited input ranges, 32 bit fixed-point implementations provide better precision than single floating-point precision because single precision has to accommodate a larger dynamic range which reduces the number of bits available for the mantissa. That said, fixed-point implementations run slower, at least on the JVM, than the more precise double floating-point arithmetic with its dedicated hardware support. However, the choice for fixed-point rather than floating-point may be also due to this hardware being unavailable. Our tool can thus support a wide variety of applications with different requirements. We also note that across the three (not specially selected) benchmarks, the results are very consistent and we expect similar behavior for other applications as well.

8.2 Evaluating Effectiveness on Nonlinear Expressions

Range computation Stepwise estimation of errors crucially depends on the estimate of the ranges of variables. The strength of using a constraint solver such as Z3 is that it can perform such estimation while taking into account the precise dependencies between variables in preconditions and path conditions. Table 2 compares results of our range computation procedure described in Section 6 against ranges obtained with standard interval arithmetic. Interval arithmetic is one of the methods used for step-wise range estimation; an alternative being affine arithmetic, which we found to give more pessimistic results in our experiments. We believe that this is due to imprecision in computing nonlinear operations. Note, however, that we still use affine arithmetic to estimate errors given the computed ranges.

For our range computation, we set the default precision threshold to 1e−10 and maximum number of iterations for the binary search to 50. To obtain an idea about the true ranges of our functions, we have also computed a lower bound on the range using simulations with 10^7 random inputs and with exact rational arithmetic evaluation of expressions. We observe that our range computation can significantly improve over standard interval bounds. The jetEngine benchmark is a notable example, where interval arithmetic yields the bound $[-\infty, \infty]$, but our procedure can still provide bounds that are quite close to the true range.

Error computation Table 3 compares uncertainties computed by our tool against maximum uncertainties obtained through extensive simulation with 10^7 random inputs for different precisions. To obtain (under-approximations of) error bounds, we ran the simulation in parallel with rational and their corresponding floating-point or fixed-point values and obtained the error by taking the difference in the result. Selected benchmarks, marked with (*), also have added uncertainties on the input parameters. To our knowledge this is the first quantitative comparison of an error computation precision with (an approximation) of the true errors on such benchmarks. Our computed uncertainties are mostly within about an order and many times even closer to the under-approximation of the true errors provided by simulation. In the case of the jetEngine$*$ benchmark, we believe that the imprecision is mainly due to its complexity and subsequent failures of Z3 in the range computation. The values in parentheses in the second column indicate errors computed if ranges at each arithmetic operation are computed using interval arithmetic alone. While we have not attempted to improve the affine arithmetic-based error computation from [15], we can see that in some cases a more precise range computation can gain us improvements. The full effect of the imprecision of standard range computation appears when, due to this imprecision, we obtain possible errors such as division-by-zero or square root of a negative number errors. The first case happens in the case of the non-linear jetEngine benchmark, so with interval arithmetic alone we would therefore not obtain any meaningful result. Similarly, for the triangle example from Section 2, without being able to constrain the inputs to form valid triangles, we cannot compute any error bound, because the radicand becomes possibly negative.

Table 4 presents another relevant experiment, evaluating the ability to use additional constraints during our range computation. For this experiment, we use double precision and the triangle example from Section 2 with additional constraints allowing increasingly flat triangles by setting the threshold on line 12 ($a + b > c + 1e{-}6$) to the different values given in the first column. As the triangles become flatter, we observe an expected increase in uncertainty on the output since the formula becomes more prone to roundoff errors. At threshold 1e−10 our range computation fails to provide the necessary precision and the radicand becomes possibly negative. Using our tool, the developer can therefore go beyond rules of thumb and informal estimates and be confident that the computed area is accurate up to seven decimal digits even for triangles that whose difference $a + b - c$ is as small as 10^{-9}.

Compilation Running Time Running times for compilation are below seven seconds for all benchmarks from Table 3 and for the sine example from Figure 4, except for jetEngine which runs in about two minutes due to timeouts from Z3 for some intermediate ranges. Examples that require computation of the path error are much more computationally challenging: verifying the postcondi-

Benchmark	Our error (IA only)	Simulated error
doppler1* (dbl)	2.36e-6	5.97e-7
doppler1 (dbl)	4.92e-13 (4.95e-13)	7.11e-14
doppler2 (dbl)	1.29e-12	1.14e-13
doppler2 (32bit)	8.67e-6 (8.79e-6)	1.57e-6
doppler3 (dbl)	2.03e-13 (2.05e-13)	4.27e-14
rigidBody1* (dbl)	9.21e-7	8.24e-7
rigidBody1 (dbl)	5.08e-13	2.28e-13
rigidBody2 (dbl)	6.48e-11	2.19e-11
jetEngine* (dbl)	0.15 (-)	3.58e-5
jetEngine (dbl)	1.62e-8 (-)	5.46e-12
jetEngine (32bit)	0.0972 (-)	1.77e-4
turbine1* (dbl)	4.86e-6	3.71e-7
turbine1 (dbl)	1.25e-13 (1.38e-13)	1.07e-14
turbine1 (16bit)	0.0487 (0.0522)	0.00623
turbine2 (dbl)	1.76e-13 (1.96e-13)	1.43e-14
turbine3 (dbl)	8.50e-14 (9.47e-14)	5.33e-15
verhulst* (dbl)	2.82e-4	2.40e-4
verhulst (dbl)	6.82e-16	2.23e-16
predatorPrey* (dbl)	9.22e-5	8.61e-5
predatorPrey (dbl)	2.94e-16 (2.96e-16)	1.12e-16
carbonGas* (dbl)	2114297.84	168874.70
carbonGas (dbl)	4.64e-8 (5.04e-8)	3.73e-9
Sine (dbl)	9.57e-16 (1.46e-15)	4.45e-16
Sine (fl)	1.03e-6 (1.57e-6)	1.79e-7
Sine (16bit)	2.87e-4 (6.79e-4)	1.55e-4
Sqrt (dbl)	8.41e-16 (8.87e-16)	4.45e-16
Sqrt (fl)	9.03e-7 (9.52e-7)	2.45e-7
Sqrt (16bit)	5.97e-4	1.58e-4
Sine, order 3 (fl)	1.19e-6 (1.55e-6)	2.12e-7
Sine, order 3 (dbl)	1.11e-15 (1.44e-15)	3.34e-16

Table 3. Comparison of errors computed with our procedure against simulated errors. Simulations were performed with 10^7 random inputs. (*) indicates that inputs have external uncertainties associated. dbl and fl denotes double and single floating-point precision respectively, 32 or 16 bit is the bit-width of the fixed-point implementation.

Benchmark	Area	Max. abs. error
triangle1 (0.1)	[0.29432, 35.0741]	2.72e-11
triangle2 (1e-2)	[0.099375, 35.0741]	8.04e-11
triangle3 (1e-3)	[3.16031e-2, 35.0741]	2.53e-10
triangle4 (1e-4)	[9.9993e-3, 35.0741]	7.99e-10
triangle5 (1e-5)	[3.1622e-3, 35.0741]	2.53e-9
triangle6 (1e-6)	[9.9988e-4, 35.0741]	7.99e-9
triangle7 (1e-7)	[3.1567e-4, 35.0741]	2.54e-8
triangle8 (1e-8)	[9.8888e-5, 35.0741]	8.08e-8
triangle9 (1e-9)	[3.0517e-5, 35.0741]	2.62e-7
triangle10 (1e-10)	-	-

Table 4. Area computed and error on the result for increasingly flat triangles. All values are rounded outwards. Interval arithmetic alone fails to provide any result.

```
   def cav10(x: Real): Real = {
2    require(0 < x && x < 10)
     if (x∗x − x >= 0)
4      x/10
     else
6      x∗x + 2
   } ensuring(res ⇒ 0 <= res && res <= 3.0 && res +/− 3.0)
8
   def squareRoot3(x: Real): Real = {
10   require(0 < x && x < 10 && x +/− 1e−10 )
     if (x < 1e−5)
12     1 + 0.5 ∗ x
     else
14     sqrt(1 + x)
   } ensuring(res ⇒ res +/− 1e−10)
16
   def smartRoot(a: Real, b: Real, c: Real): Real = {
18   require(3 <= a && a <= 3 && 3.5 <= b && b <= 3.5 &&
       −2 < c && c < 2 && b∗b − a ∗ c ∗ 4.0 > 0.1)
20
     val discr = b∗b − a ∗ c ∗ 4.0
22   if(b∗b − a∗c > 10.0) {
       if(b > 0.0) c ∗ 2.0 /(−b − sqrt(discr))
24     else if(b < 0.0) (−b + sqrt(discr))/(a ∗ 2.0)
       else (−b + sqrt(discr))/(a ∗ 2.0)
26   } else {
       (−b + sqrt(discr))/(a ∗ 2.0)
28   }
   } ensuring(res ⇒ res +/− 6e−15)
```

Figure 9. Path error computation examples.

tion on the main function from the initial example takes about one minute. Running times depend highly on whether Z3 fails to tighten intermediate ranges and the timeout used for Z3. Our default setting is one second; we did not find much improvement in the success rate above this threshold.

8.3 Evaluating Errors across Program Paths

Figure 9 presents several examples to evaluate our error computation procedure across different paths from Section 7. The first method cav10 [26] has been used before as a benchmark function for computing the output range. Our tool can verify the given postcondition immediately. Note that the error on the result is actually as large as the result itself, since the method is non-continuous, an aspect that has been ignored in previous work, but that our tool detects automatically. The method squareRoot3 is also a non-continuous function that computes the square root of $1+x$ using an approximation for small values and the regular library method otherwise. Note the additional uncertainty on the input, which could occur for instance if this method is used in an embedded controller. Our tool can verify the given specification. If we change the condition on line 10 to x < 1e−4 however, verification fails. In this fashion, we can use our tool to determine the appropriate branch condition to meet the precision requirement. These two examples verify in under 5 seconds. Finally, the smartRoot method computes one root of a quadratic equation using the well-known more precise method from [27]. Our tool compares the values error across different paths in real and finite-precision implementation and succeeds in verifying the postcondition under 25s.

9. Related work

Current approaches for verifying finite-precision code include abstract interpretation, interactive theorem proving and decision procedures, which we survey in this section. We are not aware of work that would automatically integrate reasoning about uncertainties into a programming model.

Abstract interpretation (AI) Abstract domains that are sound with respect to floating-point computations can prove bounds on the ranges of variables [8, 14, 24, 32, 42]. The only work in this area that can also quantify roundoff errors is the tool Fluctuat [21, 28]. These techniques use interval or affine arithmetic and together with the required join and meet operations may yield too pessimistic results. [47] improves the precision of Fluctuat by refining the input domains with a constraint solver. Our approach can be viewed as approaching the problem from a different end, starting with an exact constraint and then using approximation until the solver succeeds. Unlike AI tools in general, our system currently does not perform widening to ensure convergence. If the user can provide inductive postconditions, then we can still prove the code correct, but we do not in general discover these postconditions ourselves. Our focus is on proving precise bounds on the ranges in the presence of nonlinear computations and the quantification of roundoff errors and other uncertainties.

Theorem proving The Gappa tool [18, 39] generates a proof checkable by the interactive theorem prover Coq from source code with specifications. It can reason about properties that can be reduced to reasoning about ranges and errors, but targets very precise properties of specialized functions, such as software implementations of elementary functions. The specification itself requires expertize and the proofs human intervention. A similar approach is taken by [4] which generate verification conditions that are discharged by various theorem provers. Harisson has also done significant work on proving floating-point programs in the HOL Light theorem prover [30]. Our approach makes a different compromise on the precision vs. automation tradeoff, by being less precise, but automatic. The Gappa and interactive theorem provers can be used as complements to our tool: if our tool detects that more precision is needed, interactive tools can be employed by an expert user on selected methods; the results can then used by our tool in the context of the overall program.

Range computation The Gappa tool and most constraint solvers internally use interval arithmetic for sound range computations, whose limitations are well-known. [23] describes an arithmetic based on function enclosures and [41] use an arithmetic based on taylor series as an alternative. This approach is useful when checking a constraint, but is not suitable for a *forward computation* of ranges and errors.

Decision procedures An alternative approach to verification via range computation are floating-point decision procedures. Bit-precise constraints, however, become very large quickly. [11] addresses this problem by using a combination of over- and under-approximations. [29] present an alternative approach in combining interval constraint solving with a CDCL algorithm and [25] is a decision procedure for nonlinear real arithmetic combining interval constraint solving with an SMT solver for linear arithmetic.[49] formalizes the floating-points for the SMT-LIB format. While these approach can check ranges on numeric variables, they do not handle roundoff errors or other uncertainties and cannot compute specifications automatically. [46] use a floating-point decision procedure to detect stability issues and while their approach provides witnesses of instability if such exist, it is not able to prove sound error bounds with respect to a real-valued semantics. [3] prove fixed-point constraints with a combination of bit vectors and reals, but such an encoding is only possible for fixed-points and less efficient than reals alone. An alternative to our approach is using linear approximations to solve polynomial constraints [10]. We believe that such advances are largely orthogonal to our use of range arithmetic and complement each other.

Testing Symbolic execution is a well-known technique for generating test inputs. [9] use a combination of meta-heuristic search and interval constraint solving to solve the floating-point constraints that arise, whereas [37] combine random search and evolutionary techniques. [52] test numerical code for precision by perturbing low-order bits of values and rewriting expressions. The idea is to exaggerate initial errors and thus make imprecisions more visible. Probabilistic arithmetic [50] is a similar approach but it does the perturbation by using different rounding modes. [6] also propose a testing produce to detect accuracy problems by instrumenting code to perform a higher-precision computation side by side with the regular computations. While these approaches are sound with respect to floating-point arithmetic, they only generate or can check individual inputs and are thus not able to verify or compute output ranges or their roundoff errors.

Robustness analysis [31] combines abstract interpretation with model checking to check programs for stability by tracking the evolution of the width of the interval representing a single input. [40] use concolic execution to find inputs which, given maximum deviations on inputs, maximize the deviation on the outputs. These two works however, use a testing approach and cannot provide sound guarantees. [12] presents a framework for continuity analysis of programs along the mathematical $\epsilon - \delta$ definition of continuity and [13] builds on this work and presents a sound robustness analysis. This framework provides a syntactic proof of robustness for programs over reals and thus does not consider floating-points. Our approach describes a quantitative measure of robustness for nonlinear programs with floating-point numbers and other uncertainties, and we believe that it can complement the cited framework.

10. Conclusion

We have presented a programming model for numerical programs that decouples the mathematical problem description from its realization in finite precision. The model uses a **Real** data type that corresponds to mathematical real numbers. The developer specifies the program using reals and indicates the target precision; the compiler chooses a finite-precision representation while checking that the desired precision targets are met. We have described the soundness criteria by translating programs with precision requirements into verification conditions over mathematical reals. The resulting verification conditions, while a natural description of the problem being solved, are difficult to solve using a state-of-the art SMT solver Z3. We therefore developed an algorithm that combines SMT solving with range computation. Our notion of soundness incorporates full input/output behavior of functions, taking into account that, due to conditionals, small differences in values can lead to different paths being taken in the program. For such cases our approach estimates a sound upper bound on the total error of the computation.

We have evaluated our techniques on a number of benchmarks from the literature, including benchmarks from physics, biology, chemistry, and control systems. We have found that invocation of an SMT solver alone is not sufficient to handle these benchmarks due to scalability issues, whereas the use of range arithmetic by itself is not precise enough. By combining these two techniques we were able to show that a finite-precision version of the code conforms to the real-valued version with reasonable precision requirements.

We believe that our results indicate that it is reasonable to introduce **Real**s as a data type, following a list of previously introduced mathematical abstractions in programming languages, such as unbounded integers, rationals, and algebraic data types. The feasibility of verified compilation of our benchmarks suggests that it is realistic to decouple the verification of executable mathematical models over reals from their sound compilation. We therefore expect that this methodology will help advance rigorous formal verification of numerical software and enable us to focus more on high-level correctness properties as opposed to run-time errors alone. Furthermore, we expect that having real numbers as a data type facilitates automatic reordering of finite-precision computations [17] as well as high-level optimizations such as replacing one version of a numerical algorithm with another to achieve the desired combination of efficiency and rigorous worst-case bounds on precision.

References

[1] Caliper open source framework. http://code.google.com/p/caliper/.

[2] A. Anta and P. Tabuada. To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems. *IEEE Transactions on Automatic Control*, 55(9), 2010.

[3] A. Anta, R. Majumdar, I. Saha, and P. Tabuada. Automatic Verification of Control System Implementations. In *EMSOFT*, 2010.

[4] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In *IJCAR*, 2010.

[5] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. C++/Fortran-90 double-double and quad-double package. http://crd-legacy.lbl.gov/~dhbailey/mpdist/, 2013.

[6] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *PLDI*, 2012.

[7] R. Blanc, E. Kneuss, V. Kuncak, and P. Suter. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Scala Workshop*, 2013.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[9] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu. Symbolic Execution with Interval Solving and Meta-heuristic Search. In *ICST*, 2012.

[10] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Sat modulo linear arithmetic for solving polynomial constraints. *J. Automated Reasoning*, 48(1), 2012.

[11] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*, pages 69–76, 2009.

[12] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[13] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. In *ESEC/FSE*, 2011.

[14] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval Polyhedra: An Abstract Domain to Infer Interval Linear Relationships. In *SAS*, 2009.

[15] E. Darulova and V. Kuncak. Trustworthy Numerical Computation in Scala. In *OOPSLA*, 2011.

[16] E. Darulova and V. Kuncak. Certifying Solutions for Numerical Constraints. In *RV*, 2012.

[17] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha. Synthesis of Fixed-Point Programs. In *EMSOFT*, 2013.

[18] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. *IEEE Trans. Comput.*, 2011.

[19] L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. IMPA/CNPq, Brazil, 1997.

[20] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[21] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Vedrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, 2009.

[22] V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric Bounds Analysis with Conflict-driven Learning. In *TACAS*, 2012.

[23] J. A. Duracz and M. Konecny. Polynomial function enclosures and floating point software verification. In *CFV/IJCAR*, 2008.

[24] J. Feret. Static Analysis of Digital Filters. In *ESOP*, 2004.

[25] S. Gao, M. Ganai, F. Ivancic, A. Gupta, S. Sankaranarayanan, and E. Clarke. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems. In *FMCAD*, 2010.

[26] K. Ghorbal, E. Goubault, and S. Putot. A Logical Product Approach to Zonotope Intersection. In *CAV*, 2010.

[27] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1), 1991.

[28] E. Goubault, S. Putot, and F. Védrine. Modular static analysis with zonotopes. In *SAS*, 2012.

[29] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, 2012.

[30] J. Harrison. Floating-Point Verification using Theorem Proving. In *Formal Methods for Hardware Verification, SFM*, 2006.

[31] F. Ivancic, M. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, 2010.

[32] B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, 2009.

[33] J. Jiang, W. Luk, and D. Rueckert. FPGA-Based Computation of Free-Form Deformations. In *FPL*, 2003.

[34] D. Jovanović and L. de Moura. Solving Non-linear Arithmetic. In *IJCAR 2012*, 2012.

[35] W. Kahan. Miscalculating Area and Angles of a Needle-like Triangle. Technical report, University of California Berkeley, 2000.

[36] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *PLDI*, 2012.

[37] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution. In *Testing Software and Systems*. Springer Berlin / Heidelberg, 2010.

[38] X. Leroy. Verified squared: does critical software deserve verified tools? In *POPL*, 2011.

[39] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan. Towards program optimization through automated analysis of numerical precision. In *CGO*, 2010.

[40] R. Majumdar, I. Saha, and Z. Wang. Systematic Testing for Control Applications. In *MEMOCODE*, 2010.

[41] K. Makino and M. Berz. Taylor Models and Other Validated Functional Inclusion Methods. *International Journal of Pure and Applied Mathematics*, 4, 2003.

[42] A. Miné. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *ESOP*, 2004.

[43] R. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[44] J. D. Murray. *Mathematical Biology*. Springer, 2002.

[45] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, 2008.

[46] G. Paganelli and W. Ahrendt. Verifying (In-)Stability in Floating-point Programs by Increasing Precision, using SMT Solving. In *SYNASC*, 2013.

[47] O. Ponsini, C. Michel, and M. Rueher. Refining Abstract Interpretation Based Value Analysis with Constraint Programming Techniques. In *CP*, 2012.

[48] A. Quarteroni, F. Saleri, and P. Gervasio. *Scientific Computing with MATLAB and Octave*. Springer, 3rd edition, 2010.

[49] P. Rümmer and T. Wahl. An SMT-LIB Theory of Binary Floating-Point Arithmetic. In *Informal proceedings of SMT at FLoC*, 2010.

[50] N. Scott, F. Jézéquel, C. Denis, and J.-M. Chesneaux. Numerical 'health check' for scientific codes: the CADNA approach. *Computer Physics Communications*, 2007.

[51] I. C. Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008.

[52] E. Tang, E. Barr, X. Li, and Z. Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *ISSTA*, 2010.

[53] E. M. Westbrook and S. Chaudhuri. A Semantics for Approximate Program Transformations. *CoRR*, 2013.

[54] C. Woodford and C. Phillips. *Numerical Methods with Worked Examples*, volume 2nd. Springer, 2012.