Software Model Checking the Precision of Floating-Point Programs

Franjo Ivančić, Malay K. Ganai, Sriram Sankaranarayanan, and Aarti Gupta
NEC Laboratories America
4 Independence Way, Suite 200, Princeton, New Jersey 08540

***Abstract*—Software model checking has recently been successful in discovering bugs in production software. Most tools have targeted heap related programming mistakes and control-heavy programs. However, real-time and embedded controllers implemented in software are susceptible to computational numeric instabilities. In this work, we target numerical programs implemented using the IEEE 754 floating-point standard, and the precision loss incurred in such programs. There have been techniques that handle analysis of such numerical programs using abstract interpretation based techniques. We use bounded model checking (BMC) based on Satisfiability Modulo Theory (SMT) solvers to analyze programs with floating point operations. We generate a mixed integral-real model that is then analyzed by two backend model checkers.**

I. Introduction

With the growth of multi-core processing and concurrent programming in many key computing segments (mobile, server, gaming), there is a need for effective development and verification technologies for concurrent multi-threaded programs. At the same time, due to the ubiquitous availability of real-time and cyber-systems that interact with physical environments, there is a great need to develop technologies that target the whole system. Analyzing software for its correctness is a key step in guaranteeing safety of many important real-time and embedded devices, such as medical devices, automobiles or airplanes.

Recently, there has been extensive research on model checking software programs, including [5], [10], [21], [22], [25] and many others. All of these techniques try to cover as many different language features as possible, but lately the focus has been on memory correctness issues due to intricate use of pointer indirections, for example. The only tools known to us that handle floating-point operations are based on the CPROVER infrastructure (CBMC [8] and SATABS [9]). The tool generates a bit-blasted formula for floating-point computations that is translated directly to a SAT solver in the backend. The created formula is inherently very precise but at the same time it does not provide a scalable solution to analyzing programs with floating-point operations. Additionally, in an approach that models floating point operations exactly, it is not possible to discern whether a particular operation causes significant precision loss. However, the precision loss due to such numerically unstable implementations are often overlooked by control engineers. Finally, it should be noted that there are no specific floating-point related checks that are performed in the CPROVER infrastructure.

On the other hand, there are several tools based on abstract interpretation that target the ever growing embedded software domain. These tools focus mostly on floating-point semantics, given their prevalence and importance for the safety of real-time and embedded software such as used in medical devices, cars, airplanes and so on. These tools include `ASTRÉE` [11], `FLUCTUAT` [20], and `PolySpace` [28]. These tools provide scalable analysis techniques based on abstract interpretation by limiting precision in certain cases, for example due to widening of loops. Finally, abstract interpretation based tools lack the capability to generate concrete counterexamples which is a crucial benefit of model checkers.

A. Motivating Example

We introduce a small instructive example, adapted from [19], that shows a program with significant precision loss. The function CTRLTEST first computes the expression x= $\frac{c_1 b_2 - c_2 b_1}{a_1 b_2 - a_2 b_1}$, and returns TRUE if the answer is non-negative. For the input $a_1 = 37639840, a_2 = 29180479, b_1 = -46099201, b_2 = -35738642, c_1 = 0$, and $c_2 = 1$ using single-precision floating-point arithmetic, the program computes the value x$= 0.343466$ on a PC running Linux using the gcc compiler in default rounding mode. In [19], the author observes that the same computation results in x=1046769994 on an UltraSparc architecture. However, mathematically speaking, the computation of x should result in x$=-46099201$.

Motivating Example
Output: returns TRUE, if $\frac{c_1 b_2 - c_2 b_1}{a_1 b_2 - a_2 b_1}$ is non-negative; FALSE otherwise.

```
1: procedure CTRLTEST(floats a1, a2, b1, b2, c1, c2)
2:     float x = (c1*b2-c2*b1)/(a1*b2-a2*b1);
3:     if x ≥ 0.0f then return TRUE;
4:     else return FALSE;
5:     end if
6: end procedure
```

The problem observed in this example is known to be due to *cancelation* effects. Since the sign determines the output of the function CTRLTEST, which may lead to a downstream choice of a different controller being activated, such an unstable computation may result in compromised safety of embedded and real-time devices. It should be noted that CBMC is able to analyze this small example and produces a value x$= 0.343466$. However, CBMC is not able to discover that

this computation is numerically unstable due to its inherent bit-precise reasoning of the computation. Furthermore, the provided answer relies on a particular rounding mode that is only under control of the program under analysis. For other instructive examples showcasing numerically stable and unstable algorithms, the reader is referred to [19].

B. The F-SOFT Platform

The F-SOFT tool provides a combined infrastructure that builds a single model that is utilized by both abstract interpretation techniques [4], [30], and model checking techniques (see [17] for a description of the synergies between the two techniques). The tool analyzes source code written in C or C++ for user-specified properties, user-provided interface contracts, or standard properties such as buffer overflows, pointer validity, string operations, etc. Extensions of F-SOFT to concurrent programs are described elsewhere [26], and will not be discussed further here. The central role abstract interpretation plays in F-SOFT is to eliminate easy proofs and allow model simplifications (such as program slicing [27] for example) to generate small enough models that can be passed on to the model checker in the backend. F-SOFT supports both unbounded model checking as well as bounded model checking (BMC) techniques, where BMC can utilize either a SAT-solver [6] or an SMT-solver [3] for the analysis.

In this paper we describe the improvements to the F-SOFT infrastructure to reason precisely about floating-point operations using an SMT-solving backend. Earlier, floating-point operations were treated as nondeterministic operations, sometimes resulting in too many incorrect warnings. Given the essential safety concerns related to embedded devices, a more precise while scalable analysis is essential.

The verification model generated by F-SOFT is a control-flow graph representation over integral type variables combined with variables in the real domain.[1] Floating-point variables in the source code are modeled by a lowering mechanism that utilizes additional monitoring variables. These monitoring variables contain variables in the reals, as well as variables of a small enumerated type denoting whether the floating-point variable is `NaN`, some infinity denoted as `Inf`, or represents an actual number. The monitoring variables that are of type real represent a bounding interval in the model for the associated variable, as long as the type of the variable is *numeric*, i.e., it is not `NaN` or some `Inf`.

C. Paper Contributions and Outline

This paper presents a software verification approach to the analysis for numerically unstable computations given IEEE 754 floating-point semantics that is based on creating a model and utilizes model checking for the analysis of the model. We present a model that incorporates modeling of all relevant arithmetic operations and casting constructs between floating-point types and other program types. We also model soundly arbitrary rounding modes as well as sub-normal numbers as defined in the IEEE 754 standard. Finally, we implemented the model generation in the F-SOFT tool and present experimental results on a variety of benchmarks that can be scaled to create different sizes. These experiments show that the performance of the model checkers can be significantly improved by providing a priori computed invariants to simplify the models before analyzing them with a model checker.

The next section introduces the IEEE 754 floating-point standard. Section III describes the modeling of C programs with floating-point operations inside the F-SOFT tool. We present experimental results using the proposed framework with two backend model checkers in section IV. Finally, section V concludes the paper with some final remarks and an overview of potential further research directions.

II. THE IEEE 754 FLOATING-POINT STANDARD

We analyze source code with respect to the binary formats of the new IEEE 754-2008 standard [24], which are largely based on the IEEE 754-1985 norm. The general layout of a floating-point number is in *sign-magnitude form*, where the most significant bit is a *sign bit*, the exponent is stored as a *biased exponent*, and the fraction is the *significand* stored without the most significant bit (see Fig. 1). The exponent is biased by $(2^{e-1}) - 1$, where e is the number of bits used for the exponent field. For example, to represent a number which has exponent of 17 in an exponent field 8 bits wide, we store 144 since $17 + (2^{8-1}) - 1 = 144$. In most cases, as mentioned above, the most significant bit of the significand is assumed to be 1 and is not stored. This case occurs when the biased exponent η is in the range $0 < \eta < 2^{e-1}$, and the numbers so represented are called *normalized numbers*. If the biased exponent η is 0 and the fraction is not 0, the most significant bit of the significand is implied to be 0, and the number is said to be *de-normalized* or *sub-normal*. The remaining special cases are:

- If the biased exponent is 0 and the fraction is 0, the number is ± 0 (depending on the sign bit).[2]
- If the biased exponent equals 2^{e-1} and the fraction is 0, the number is $\pm\infty$ (depending on the sign bit), which is denoted as `Inf` or $-$`Inf` here, and
- if the biased exponent equals 2^{e-1} and the fraction is not 0, the number represents the special floating-point value called *not a number* (`NaN`).

A. Floating-point formats

For ease of presentation of the IEEE standard in this section, we focus on only two floating-point formats defined in the standard: single-precision (specified using the keyword `float` in C/C++) and double-precision (specified using the keyword `double` in C/C++). Single-precision defines that a floating-point number is represented using 32 bits, of which $e = 8$ are used for the exponent and 23 bits for the fraction. Figure 1 shows the single-precision layout of the standard. The smallest

[1] For further details on F-SOFT that discuss translation from programs with structures to integer-based programs only, see [25].

[2] Note that the standard defines two zeros, namely 0 and -0. The two numbers behave similarly with a few differences. For example, dividing a positive number by 0 results in ∞, whereas dividing a positive number by -0 results in $-\infty$.

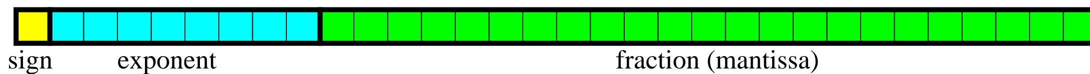

Fig. 1. General layout of a floating-point number

positive normalized number representable thus is 2^{-126} which is about $1.18 \cdot 10^{-38}$, while the largest number is $(2-2^{23}) \cdot 2^{127}$ which is about $3.4 \cdot 10^{38}$. For double-precision, on the other hand, the standard prescribes the use of 64 bits to store numbers, of which $e = 11$ represent the biased exponent, and 52 bits are used for the fraction. The largest number representable in double-precision is about $1.8 \cdot 10^{308}$.

B. Rounding

The IEEE754-2008 standard defines five different rounding modes for each floating-point operation. There are two modes that *round to nearest* neighboring floating point number, where a bias can be set to *even numbers* or *away from zero* when the operation lands exactly midway between two representable floating point numbers. The other three rounding modes are *rounding towards zero*, *rounding towards* ∞ and *rounding towards* $-\infty$. The standard defines that every arithmetic operation be calculated as precisely as possible before rounding it using the current rounding mode. Computations are thus performed using longer bit-lengths and are truncated when storing the results after rounding only. While the absolute error may be large for large absolute values, the maximum relative error due to operations and rounding thus is constant for values resulting in the normalized number range. The relative error for normalized numbers is 2^{-23} in single-precision and 2^{-52} for double-precision.

C. Sub-normal numbers

To provide *gradual underflow* for very small numbers in absolute terms, the standard introduced denormalized or sub-normal numbers. These numbers lie between the smallest positive normal number and zero, and their negative versions. This feature is meant to provide a slowing of the precision loss due to *cancelation effects* around zero. The main advantage of defining sub-normal numbers in the standard is that it guarantees that two nearby but different floating-point numbers always have a non-zero distance. Hence, any subtraction of two nearby but different floating-point numbers is guaranteed to be non-zero, which cannot be guaranteed without sub-normal numbers. However, it should be noted that operations that result in numbers in the sub-normal number range can have very large relative errors.

D. Operations

The standard defines many details about the precision, expected results and exception handling for a variety of operations such as arithmetic operations (add, subtract, multiply, divide, square root, ...), casting conversions (between formats, to and from integral types, ...), comparisons and total ordering, classification and testing for `NaN`, and many more. In this work, we will focus on arithmetic operations and casting operations in particular, using the rounding precision for operations prescribed by the standard.

III. Modeling Programs with Floating-Points

In this section, we review the software modeling in F-Soft [25] relevant to the automatic construction of a symbolic model for arbitrary C/C++ programs. F-Soft is a tool for analyzing safety properties in C/C++ programs. A large set of programming bugs, such as array bound violations, use of uninitialized variables, memory leaks, division by zero, etc. can be formulated into reachability problems by automatically adding suitable property monitors to the given program.

A. Software Modeling in F-Soft

F-Soft begins with a program in full-fledged C/C++ and applies a series of source-to-source transformations into smaller subsets of C, until the program state is represented as a collection of simple scalar variables and each program step is represented as a set of parallel assignments to these variables. We use a control-flow graph (CFG) representation as an intermediate representation. Below are details relevant to the construction of a symbolic model (for a comprehensive description of the transformations, please refer to [25]), with emphasis on issues intrinsic to C:

Pointer and Memory Modeling. One difficulty in modeling C programs lies in modeling indirect memory accesses via pointers, such as `x=*(p+i)` and `q[j]=y`. We replace all indirect accesses with equivalent expressions involving only direct variable accesses, by introducing appropriate conditional expressions as described below.

- To facilitate the modeling of pointer arithmetic, we build an internal memory representation of the program by assigning to each variable a unique natural number representing its memory address.
- We perform a points-to analysis [23] to determine, for each indirect memory access, the set of variables that may be accessed. If a pointer can point to a set of variables at a given program location, we rewrite a pointer read as a conditional assignment expression using the numeric memory addresses assigned to the variables.
- For indirect reads via pointers, we adopt an approach from hardware synthesis [31] and for each pointer variable `p` create a new variable `STAR_p` representing the current value of `*p`. Each read of `*p` is then rewritten as simply a read of `STAR_p`. (Reads of the form `*(p+i)` continue to be handled as described earlier.) To keep `STAR_p` up-to-date, after each assignment `p=q` we add an inferred assignment `STAR_p = STAR_q`. Furthermore, we need to add aliasing assignments to the model that keep `STAR_p` up-to-date, when the value may have

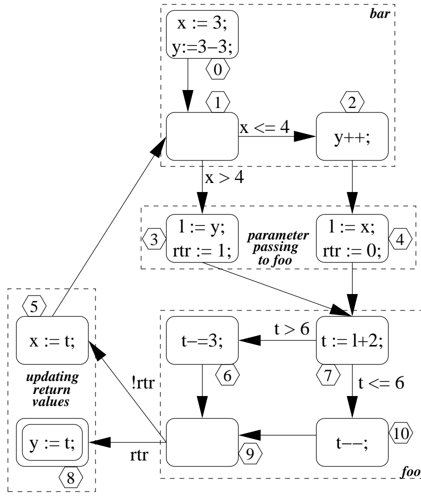

Fig. 2. Control flow graph

been changed by an assignment through `*q` or some other variable in `p`'s points-to set.

Unbounded Data, Recursion and Function Calls. The C language specification does not bound heap or stack size, but our focus is on generating a finite state model. Therefore, we model the heap as a finite array, adding a simple implementation of `malloc()` that returns pointers into this array. We also add a bounded depth stack as another global array in order to handle bounded recursion, only if required, along with code to save and restore local state for recursive functions.

An example of our modeling is shown in Figure 2, which shows the computed CFG for the following simple C code:

```
int foo(int s){        void bar() {
  int t=s+2;             int x = 3, y = x - 3;
  if (t>6)               while ( x <= 4) {
    t -= 3 ;                y++ ;
  else                      x = foo(x);
    t--;                 }
  return t;              y = foo(y);
}                      }
```

Each basic block is identified by a unique number shown inside the hexagon adjacent to the basic block. The source node of the CFG is basic block 0, while the sink node is the highlighted basic block 8. The example in Fig. 2 pictorially shows how non-recursive function calls are included in the control flow of the calling function. A preprocessing analysis determines that function `foo` is not called in any recursive manner. The two return points are recorded by an encoding that passes a unique return location as a special parameter using the variable `rtr`. As shown in Figure 2, F-SOFT builds a model of the software that includes structural features such as loops and function calls without a priori using a fixed-depth unwinding of such language features as is done in CBMC.

B. Numerical Stability Analysis of Floating-Point Operations

As mentioned before, there have been a number of proposed solutions to the numerical stability analysis of floating-point programs using abstract interpretation. The mathematical underpinnings derive from different *arithmetic models of operations*, most notably *interval arithmetic* and *affine arithmetic*. In the following, we introduce the two arithmetics.

Interval arithmetic. In order to quantify rounding errors in mathematical computations and to guarantee reliable results in numerical methods, mathematicians have studied interval arithmetic [29] (IA) for many decades. Instead of defining arithmetic operations on individual numbers, IA defines arithmetic operations on intervals on reals extended with ∞ and $-\infty$ instead. For a variable x we introduce an interval and write it as $[\underline{x}, \overline{x}]$, where $\underline{x}$ denotes the lower bound of the interval and $\overline{x}$ the upper bound. Note that the bounds can be a real or $\pm\infty$. As sample arithmetic operations consider the addition or multiplication of two intervals $[\underline{x}, \overline{x}], [\underline{y}, \overline{y}]$ which is defined as

$$[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}], \text{and}$$

$$[\underline{x}, \overline{x}] \cdot [\underline{y}, \overline{y}] = [\min(\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy}), \max(\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy})],$$

respectively. In a non-disjunctive analysis setting, division by an interval that contains zero is defined to result in the interval $[-\infty, \infty]$.

Affine arithmetic. Affine arithmetic [1] (AA) has been proposed as an improvement over IA by being able to track dependencies between variables, thus often providing less over-approximate results than IA. In AA, a quantity x is represented as a first-degree ("affine") polynomial

$$x_0 + x_1 \cdot \epsilon_1 + x_2 \cdot \epsilon_2 + \ldots + x_k \cdot \epsilon_k,$$

where $x_0, x_1, \ldots x_k$ are known real numbers, and $\epsilon_1, \epsilon_2, \ldots \epsilon_k$ are error variables, whose value is only known to be in $[-1, 1]$. Each error variable ϵ_i represents some source of uncertainty or error in the quantity x — which may come from input data uncertainty, formula truncation, or arithmetic rounding. An error variable that appears in two quantities x and y implies that their values are partially correlated.

As an example, consider $x = 3 + \epsilon_1$ and $y = 2 - \epsilon_1$. This implies that $x \in [2, 4]$ and $y \in [1, 3]$, and in IA we would find that $x + y \in [3, 7]$. However, using AA, we find that the result of $x + y$ is very close to 5; namely $5 + w \cdot \epsilon_2$, where w, ϵ_2 are introduced to model arithmetic rounding, for example, and w would be a very small quantity representing machine precision.

The abstract interpretation based tool `FLUCTUAT` [20] successfully applied AA to reason about the precision of floating-point operations. The tool generates an error term for each floating-point operation and updates the corresponding weights using widening techniques inside loops in the CFG. It has successfully been used on small, but numerically important, parts of embedded controllers and has been able to point out numerically unstable computations.

C. Modeling for Model Checking

Although our technique is largely inspired by the approach taken in `FLUCTUAT`, we have decided to implement an IA-based approach instead of an AA-based approach for scalability of model checking reasons. It is well known that

each AA operation is more expensive than each IA operation. However, AA proponents mention that for many applications the additional precision of AA allows earlier termination of an algorithm than would have been the case with IA, so that the runtime cost cannot be purely measured by comparing individual operations. While we agree with this assessment, we still believe that for model checking, the additional overhead to model AA may be overly expensive.

As a first concern, consider the number of state variables in a model which is often an important consideration for model checking. In an IA-based model, we require at least $2|V_F|$ variables to build a model for intervals for a set of floating point variables V_F. However, when building a model using AA, we would require at least $|F|$ variables to model the weight of each operation of the set of floating point operations F. This could be simplified by providing some kind of heuristics deciding which weights and error terms to join together dynamically. However, these heuristics are very hard to generalize for arbitrary source code. Furthermore, anecdotal experience suggests that AA has scaling issues in symbolic execution based approaches, without a carefully designed joining of error terms.

Based on this intuition, we have decided to initially pursue the IA-based approach for generation of a model for model checking purposes. We expect that the increased path-sensitivity in our application will be able to provide precision that is lost in an abstract interpretation based setting. Finally, we expect savings by combining abstract interpretation with model checking, as we have earlier seen for other properties in F-SOFT. For the abstract interpretation part of the tool flow, we expect to utilize AA based analysis.

Handling arithmetic operations. To model floating-point operations, we introduce a number of monitoring variables for each floating-point variable f. We use $\underline{\mathtt{f}}$ to represent the lower interval bound variable for f, and $\overline{\mathtt{f}}$ to represent the upper interval bound variable. The modeling variables $\underline{\mathtt{f}}$ and $\overline{\mathtt{f}}$ are semantically reals, and not floating-point variables.

To model special floating-point status flags such as NaN and $\pm$Inf, we introduce two status variables $\check{\mathtt{f}}, \hat{\mathtt{f}} \in \mathcal{F}$, where $\mathcal{F} := \{\mathtt{NaN}, \mathtt{Inf}, -\mathtt{Inf}, \mathtt{Number}\}$ represents the set of modeled floating-point status flags. The variable $\check{\mathtt{f}}$ represents the floating-point status of the lower interval bound variable $\underline{\mathtt{f}}$, and $\hat{\mathtt{f}}$ represents the floating-point status of the upper interval bound variable $\overline{\mathtt{f}}$. Note that $\mathcal{F}$ does not distinguish between zero, normalized or sub-normal floating-point numbers, which all carry the status Number.

The value of the variable $\underline{\mathtt{f}} \in \mathbb{R}$ is relevant only if the corresponding type variable $\check{\mathtt{f}} = \mathtt{Number}$. This produces the constraint

$$\check{\mathtt{f}} = \mathtt{Number} \rightarrow \mathtt{f} \geq \underline{\mathtt{f}},$$

and a corresponding constraint for the upper bound:

$$\hat{\mathtt{f}} = \mathtt{Number} \rightarrow \mathtt{f} \leq \overline{\mathtt{f}}.$$

For every arithmetic floating-point operation, such as $\mathtt{z} := \mathtt{x} +_F \mathtt{y}$, we add additional statements to the model of the program that update the related monitoring variables. In the sequel, we use the following notation for modeling rounding in the model: $\downarrow(x)$ models rounding of x towards $-\infty$, whereas $\uparrow(x)$ models rounding towards ∞ using a rounding precision that is based on the compile-time floating-point type of the expression. Furthermore, we use $\sigma, \lambda \in \mathbb{R}$ to represent the allowed bounds of a particular floating-point type; that is a value less than σ is treated as -Inf, whereas a value larger than λ is treated as Inf.

Note that the values σ, λ and the definition of $\downarrow, \uparrow$ are dependent on the actual compile-time type of the floating-point used. We currently support the following floating-point types: $\mathcal{T} := \{$float, double, long double$\}$. For the above statement, we introduce the following additional statements (written using the C ternary operator ?) in our model:

$$\begin{aligned}
\underline{\mathtt{z}} &:= \downarrow(\underline{\mathtt{x}} + \underline{\mathtt{y}}), \\
\overline{\mathtt{z}} &:= \uparrow(\overline{\mathtt{x}} + \overline{\mathtt{y}}), \\
\check{\mathtt{z}} &:= (\check{\mathtt{x}} = \mathtt{NaN} \vee \check{\mathtt{y}} = \mathtt{NaN})?\mathtt{NaN} : \\
&\quad\ (\check{\mathtt{x}} = \mathtt{Inf} \wedge \check{\mathtt{y}} = -\mathtt{Inf})?\mathtt{NaN} : \\
&\quad\ (\check{\mathtt{x}} = -\mathtt{Inf} \wedge \check{\mathtt{y}} = \mathtt{Inf})?\mathtt{NaN} : \\
&\quad\ (\check{\mathtt{x}} = \mathtt{Inf} \vee \check{\mathtt{y}} = \mathtt{Inf})?\mathtt{Inf} : \\
&\quad\ (\check{\mathtt{x}} = -\mathtt{Inf} \vee \check{\mathtt{y}} = -\mathtt{Inf})? - \mathtt{Inf} : \\
&\quad\ (\underline{\mathtt{z}} < \sigma)? - \mathtt{Inf} : (\underline{\mathtt{z}} > \lambda)?\mathtt{Inf} : \mathtt{Number},
\end{aligned}$$

and similarly for $\hat{\mathtt{z}}$.

While the update to the floating-point status variables is quite complex, it should be noted that these variables only range over $\mathcal{F}$. Often, these updates can be simplified by a priori computed interval invariants for each floating-point variable, thus allowing us to resolve many of the conditions before generating this complex update function.

Finally, note that we change the type of the original floating-point variables in the program to be of type real in the model. Thus, all variables in the model are either of an integral type or of type real.

Casting. In addition to handling arithmetic operations, we also model casting operations between floating-point types and between floating-point variables and integral type variables. The following casting operations are supported:

- Casting from lower precision floating-point types to higher precision ones, such as a cast from float to double, does not lose precision.
- Casting from higher precision types to a lower type requires rounding using $\downarrow$ for $\underline{\mathtt{f}}$ and $\uparrow$ for $\overline{\mathtt{f}}$, while it may also cause a change in the floating-point status variables from Number to $\pm$Inf.
- Casting from an integer variable to a floating-point variable requires the use of the rounding modes.
- However, casting from floating-point to some integer variable is only well defined if the variable is large enough to represent the value of the floating-point; otherwise, the result is undefined, specified as a nondeterministic value in the range of the integral type variable.

D. Modeling Rounding

This section focuses on modeling the rounding of floating-point operations, which has been denoted as $\uparrow$ and $\downarrow$ in the

previous sections. We first focus on the normalized number range first, and later on the sub-normalized number range.

Rounding normalized numbers. As mentioned above, the relative precision for normalized numbers is fixed given a particular floating-point type $t \in \mathcal{T}$. For every floating-point type $t \in \mathcal{T}$, we introduce a constant δ_t that is larger than or equal to the worst-case relative error for the floating-point type t in a normalized number range. Note that we will simply write δ when the floating-point type is known or inconsequential.

By keeping δ_t close to the maximal relative error we can avoid too many false warnings, while it may make individual computations during the analysis stage more expensive. By increasing δ_t we keep the analysis model sound, but may introduce false warnings due to an overestimation of the relative rounding error.

For the normalized number range we thus define rounding functions $\downarrow_n, \uparrow_n : \mathbb{R} \to \mathbb{R}$ as

$$\downarrow_n (x) := \begin{cases} x(1-\delta) & : \quad x \geq 0, \\ x(1+\delta) & : \quad x < 0. \end{cases} \text{, and}$$

$$\uparrow_n (x) := \begin{cases} x(1+\delta) & : \quad x \geq 0, \\ x(1-\delta) & : \quad x < 0. \end{cases} .$$

For ease of presentation, we avoid the floating-point type annotation to the rounding functions, although these rounding functions in fact are floating-point type specific.

Rounding sub-normal numbers. A relative error model of sub-normal numbers is ineffective, since the relative error can be 1 inside the sub-normal number range. Consider the fact that a number that is smaller than the smallest sub-normal number may be rounded down to 0. It would be possible to introduce various ranges of sub-normal numbers and treat them separately. Instead, we have decided to use absolute error modeling in the combined sub-normal number range for now. We choose a number ϵ_t for $t \in \mathcal{T}$ that is larger than or equal to the largest absolute error in a floating-point type specific sub-normal number range and use the absolute error for operations resulting in values in the sub-normal number range.

We define rounding functions $\uparrow_s, \downarrow_s : \mathbb{R} \to \mathbb{R}$ for the sub-normal number range as:

$$\downarrow_s (x) := x - \epsilon, \text{ and } \uparrow_s (x) := x + \epsilon.$$

The main disadvantage of this rounding model is that most operations that could result in a sub-normal number range yield intervals that include 0. This may lead to a large estimated error should this interval be used as a denominator in a division in a non-disjunctive analysis setting. As mentioned above, further splitting the sub-normal number range into regions may provide better accuracy.

Combining rounding functions. A straightforward solution to combining the rounding functions $\uparrow_n, \uparrow_s$ to yield a combined rounding function $\uparrow$ would be to compute the result of an operation, and based on the result choose which rounding function to use. This would introduce additional floating-point type specific constants delineating the boundary between normalized numbers and sub-normal numbers. In addition, each expression would require one further ITE (*if-then-else*) to denote this choice.

In order to simplify the expressions for analysis, we have chosen to define $\uparrow, \downarrow : \mathbb{R} \to \mathbb{R}$ to be

$$\downarrow := \downarrow_n + \downarrow_s, \text{ and } \uparrow := \uparrow_n + \uparrow_s .$$

Note that this is a sound modeling because we always enforce both error types although only one is applicable at each point in time. Furthermore, note that the absolute error ϵ introduces in $\uparrow_s, \downarrow_s$ is very small and is quite immaterial as long as numbers are not within close distance to the sub-normal range.

IV. EXPERIMENTS

In the previous sections we have introduced a method to transform an arbitrary C program with floating-point operations into a CFG representation that only contains variables of integral type and real variables. Based on this representation, we translate the CFG into a model for backend solvers that can reason about models that incorporate mixed integer and real constraints. This translation is based on adding a program counter variable that is modeled as an integer ranging over the (numbered) nodes of the CFG.

We generate analysis models for two backend solver infrastructures: `HySAT` [15] and FACE [18]. Both solvers perform bounded and unbounded model checking using SMT-solvers on mixed integer-real problems. Both solving architectures are in principle sound modulo implementation defects. In the following, we briefly introduce the two solvers.

A. Model Checkers

`HySAT`**.** `HySAT` is a satisfiability checker for Boolean combinations of arithmetic constraints over real- and integer-valued variables which can also be used as a bounded model checker for hybrid (discrete-continuous) systems. A peculiarity of `HySAT`, which sets it apart from many other solvers, is that it is not limited to linear arithmetic, but can also deal with non-linear constraints involving transcendental functions.

The algorithmic core of `HySAT` is the iSAT algorithm, a tight integration of recent SAT solving techniques with interval-based arithmetic constraint solving. Details about `HySAT` and the iSAT algorithm can be found in [15].[3]

`HySAT` has been evaluated mainly on models related to the verification of hybrid systems. Due to the inherent scalability problems of hybrid systems verification and the model constraints generally imposed, the application of `HySAT` to the domain of analyzing stability of floating-point computation is challenging in a number of directions: First, we generate models of programs with a relatively large number of CFG nodes. In contrast, a typical hybrid automaton has a smaller number of control nodes. Second, our models contain deeply nested ITEs that are generally not of concern in hybrid systems verification. In fact, the `HySAT` modeling language does not support ITEs directly. We have modeled ITEs using one

[3] In the following sections on experimental results, we present data for two different versions of `HySAT`. The currently publicly available official release of `HySAT` is v0.8.4. We have been provided an early new release v0.8.5β that fixed certain bugs uncovered in our experiments. Unfortunately, this version is a preliminary release that is compiled without compiler optimizations and is expected to perform about $3\times$ slower than an optimized binary.

additional Boolean variable and one additional variable of a numeric type per ITE. The additional numeric type variable can be of type real or integer based on the subexpression in question. However, this model increases the number of variables considerably due to the depth of ITE nesting.

FACE**.** FACE is our in-house satisfiability checker for Boolean combinations of arithmetic constraints over real- and integer-valued variables. It utilizes CORDIC algorithms [32], [33], [2] to linearize non-linear arithmetic constraints given an user-specified precision requirement. It also uses a normalization scheme, combined with interval bounds to obtain a linearized formula with reduced constraints, without compromising the precision requirements. On such a linearized formula, it employs a decision procedure that uses off-the-shelf SMT(LRA) (Satisfiability Modulo Theory for Linear Real Arithmetic) solvers such as Yices [14] to explore various combination of interval bounds in a refinement-based search. The decision procedure is similar to [16] which handles non-linear integer arithmetic using an iterative lazy bounding refinement algorithm build over a SMT(LIA) solver. However, it cannot be applied to solve decision problems with non-linear operations, involving transcendental and algebraic functions over reals.

FACE utilizes CORDIC algorithms to translate non-linear arithmetic into linear arithmetic given some precision requirement. These algorithms were introduced to compute transcendental and algebraic functions using only adders and shifters in a finite recursive formulation. The number of recursive steps is determined by the accuracy requirements. On the translated formula, a DPLL-style [13], [12] interval search engine is utilized to explore all combinations of interval bounds. The search engine uses a SMT(LRA) solver to check the feasibility of a particular interval bound combination in conjunction with the linearized formula in an incremental fashion. To ensure soundness of the prototype, FACE uses infinite-precision arithmetic which can cause significant slowdown when computations require complex rational representation.

B. Experiments on Motivating Example

In this section, we present experimental results for the motivating example presented in section I. We analyze the precision of the program by splitting the statements into a sequence of floating-point operations. We only present the automatic analysis results for the floating-point operations that are not trivial (such as multiplication of the precisely handled constants 0 and 1). The computation is thus split into the following sequence (after some trivial rewriting):

- `float a1b2 = a1*b2;`
- `float a2b1 = a2*b1;`
- `float denom = a2b1-a1b2;`
- `float x = b1 / denom ;`

We analyze the computation steps for stability in the sense of having a small relative error. This is evaluated by computing the maximum error for the range of the interval computed for each computation step.

The computations of `a1b2` and `a2b1` are deemed to be stable by our analysis. In this experiment we define stable to mean that the result is accurate within 0.1%. The computation time for each step is marginal using FACE, taking $0.9s$ and $1.0s$, respectively. For the computation of `denom`, the analysis finds that the result is potentially unstable due to cancelation effects. FACE reports a potential witness at depth 18 in $6.0s$. It presents a witness to the user, where the lower bound of `denom` is negative, while the upper bound is positive. The witness produces by FACE interpreted as an interval may not be maximal. In this case, FACE produced the answer that `denom` may be within $[-4.8 \times 10^{13}, 5.0 \times 10^{-38}]$.

Since `denom` is then used as a denominator in a division, the resulting status flags for `x` become $\pm$`Inf`, which is also defined to be an unstable computation. FACE finds a witness for this check at depth 23 after $22.4s$. In the witness trace for this property, the range found for `denom` is $[-8.0\times10^{12}, 2.2\times 10^{10}]$, causing the following values for the status flags of x: $\check{\texttt{x}}$=`-Inf`, $\hat{\texttt{x}}$=`Inf`.

C. Effect of Simplifying Floating-point Type Usage

In this section, we analyze the benefit of an a priori abstract interpretation stage to simplify the model before it is passed on to the model checker. In particular, to compare the benefit in terms of simplified expressions, we only simplify statements related to the floating-point status flags; i.e., a model where $\check{\texttt{f}}, \hat{\texttt{f}} \in \mathcal{F}$ are simplified by setting them to `Number`.

The example that we are analyzing is computing the sum of an array with unknown content of type `int` and unknown array length. The property that we are analyzing is to generate a sum that is a progressively larger floating-point number. By increasing the target floating-point number we generate longer witness traces. For each numeric target we create two models: The first model consists of the model as it has been described so far. The second model propagates the fact that all floating-point variables are always of status `Number` in the model thus eliminating variables and simplifying many assignments.

Note that these simplifications are valid in that this constraint is indeed an invariant of the system. However, the model checker may have to discover this fact during its analysis. Table I summarizes the results of the experiments which were performed using FACE, as well as `HySAT` v0.8.4 and v0.8.5β.

As can be seen in table I, we have generated a set of benchmarks with increasing size. Each benchmark is available in its complete format (denoted `full` in the table), as well as a simplified version (denoted `simple`) where the floating-point status variables are removed. The two-digit number in the name of the model describes the depth at which a witness is reachable in the benchmark. The various experiments were run with a one-hour time limit on Linux servers allowing 3GB of memory usage per execution run. If the analysis is aborted due to a time out or a memory out, we also provide information on the analysis performance thus far. For example, the entry `Time-Out (37.3s @ 6)` denotes that the model checker found an unsatisfiable answer at depth 6 after 37.3 seconds, while it did not finish analyzing the problem generated for depth 7 given the overall 1 hour time out. Furthermore, we write – to denote that we did not attempt an analysis for a particular benchmark.

Model name	FACE	HySAT v0.8.4	HySAT v0.8.5β
simple-12	0.3s	Mem-Out (19.1s @ 11)	Mem-Out (43.0s @ 11)
full-12	0.3s	Time-Out (37.3s @ 6)	Time-Out (84.5s @ 6)
simple-19	2.5s	Mem-Out (258.6s @ 18)	Mem-Out (582.6s @ 18)
full-19	3.0s	Time-Out (39.6s @ 6)	Time-Out (88.8s @ 6)
simple-26	16.1s	Mem-Out (259.7s @ 18)	Mem-Out (596.8s @ 18)
full-26	46.3s	Time-Out (39.1s @ 6)	Time-Out (89.1s @ 6)
simple-33	84.3s	Mem-Out (259.8s @ 18)	Mem-Out (599.1s @ 18)
full-33	64.8s	Time-Out (39.5s @ 6)	Time-Out (86.6s @ 6)
simple-40	268.1s	—	—
full-40	494.3s	—	—
simple-47	1689.6s	—	—
full-47	1639.2s	—	—
simple-54	Time-Out (527.4s @ 53)	—	—
full-54	Time-Out (1126.0s @ 53)	—	—

TABLE I
EFFECT OF FLOATING-POINT TYPE MODEL SIMPLIFICATION ON MODEL CHECKER PERFORMANCE

It should be noted that the loop-program being analyzed contains two rounding operations within each execution of the loop body. First, we cast an integer to a floating-point variable. Secondly, we add two floating-point numbers. While the program does not contain any multiplication, our modeling of the rounding modes using $\uparrow$ and $\downarrow$ introduces a number of scalar multipliers per loop iteration.

Using FACE, we were able to analyze the generated models up to depth 53 within the one hour time limit. We have observed that an unsatisfiable solution was generally found much faster in the simplified models (see for example, the performance for depth 53 in `simple-54` and `full-54`). However, there is no apparent performance trend when analyzing the final depth with a satisfiable answer. Surprisingly, in certain cases, such as `full-47` vs. `simple-47`, the model checker finds a witness trace faster for the full model.

The experimental results also show that, unfortunately, HySAT was not able to scale very well for these generated models. However, it is clear from the data that the HySAT performance is significantly better when analyzing models that are already simplified. As mentioned before, the main application domain of HySAT was the analysis of hybrid systems with generally tight bounds on variable ranges. In our example, we generate models with large ranges which seems to cause some performance issues.[4]

D. Effect of Simplifying Floating-point Assignments

In the previous section, we analyzed the benefit of an up-front abstract interpretation stage to simplify the model in terms of floating-point status flags before model checking. In this section, we further analyze the benefits that an a priori abstract interpretation can provide if we utilize it to simplify assignments to other monitoring variables in the model also. In particular, we analyze the effect of simplifying the ITEs used in assignments to $\underline{\mathtt{f}}, \overline{\mathtt{f}}$ for some program variable `f`. We are seeking the following simplifications:

[4]We have provided a sample model to the HySAT team for debugging and hope to hear back soon with further bug fixes and improvements.

- We simplify the assignments by predicting the sign of the result of a computation. This allows us to simplify $\downarrow, \uparrow$ by removing the condition used in $\downarrow_n, \uparrow_n$.
- We simplify monitoring assignments due to multiplication that contain $\min$ and $\max$ functions. Currently, $\min, \max$ are not handled by the backend model checkers and need to be a priori simplified using nested ITEs.
- We further simplify the assignments due to modeling of rounding by predicting whether the result of a computation is a sub-normal number or whether it lies in the normal number range.

We use a slightly altered program to generate a multitude of benchmarks. The program again computes the sum of an input array. The input array contains unknown positive integer values. To compute the sum of the array in terms of floating point numbers, we model a cast from `int` to `float`, as well as the rounding during each addition operation.

The property of interest in this benchmark is that the computation is stable, i.e., that the maximal relative error of the whole computation sequence is less than some threshold. We fixed the threshold to be 10% regardless of the fixed length of the input array. This implies that there are less computations that require rounding for shorter arrays thus making the property simpler to analyze for shorter arrays besides the shorter depth of the analysis. Table II summarizes the results of the experiments which were performed using FACE only. Note, that we tried to utilize HySAT also but were not able to find a satisfiable answer in any instance.

We analyzed the program for progressively larger arrays with unknown integer contents. The length of the arrays is given in column labeled `length` in table II. We formulated the property indirectly as a reachability property, and column `depth` provides the witness depth in the bounded model checker. The column `full` denotes the run-time of the complete model as discussed in this paper. The column `simplified` $\mathcal{F}$ provides the run-times for models where we provided the knowledge that each floating-point status variable is equal to `Number`. These models are equivalent to the `simple` models in table I.

length	depth	full	simplified $\mathcal{F}$	simpler updates	number range
1	28	1.2s	—	—	—
2	38	5.3s	—	—	—
3	48	Time-Out (16.1s @ 47)	16.0s	—	—
4	58	—	90.9s	—	—
5	68	—	82.0s	—	—
6	78	—	Time-Out (14.0s @ 77)	231.2s	106.5s
7	88	—	—	277.8s	57.9s
8	98	—	—	582.3s	Time-Out (124.9s @ 97)
9	108	—	—	1072.7s	151.8s
10	118	—	—	2558.1s	755.4s
11	128	—	—	2933.0s	2357.2s
12	138	—	—	1639.5s	776.7s
13	148	—	—	Time-Out (2.6s @ 145)	2298.4s
14	158	—	—	Time-Out (2215.1s @ 157)	1636.5s
15	168	—	—	—	Time-Out (4.6s @ 165)

TABLE II
EFFECT OF INVARIANT-BASED MODEL SIMPLIFICATIONS ON MODEL CHECKER PERFORMANCE

The column `simpler updates` utilizes further information about pre-computed invariants by simplifying the nested ITE structures. Finally, we give the results for a further simplification step, where we use the pre-computed invariants to choose the appropriate number range also. In this experiment, all computations resulted in numbers in the normalized number range, so we defined $\uparrow:=\uparrow_n$ and $\downarrow:=\downarrow_n$. All performed experiments were run using a one hour time-out.

The experiments clearly show that a priori computed invariants such as those generated by abstract interpretation can benefit model checker performance by allowing significant model simplifications. In our experiments, we did not utilize the user-provided invariants to analyze the property at hand, in order to perform a fair comparison of the model checker performance on the resulting models. In practice, the invariants can sometimes validate certain properties, which can be removed before using the model checker. Finally, there is one outlier in the experiments for array length 8 that needs to be further investigated.

E. Effect of Modeling Rounding Precision

In this section, we extend the example used in section IV-D, and experiment with a number of different modeling settings for the precision of the rounding δ used in the definition of $\uparrow_n$ and $\downarrow_n$. We performed a number of additional experiments with the same benchmark where the unknown array is of fixed length $n \in [5, 35]$. We used a variety of different settings for δ, which all provide sound answers to the verification problem at hand. The experiments are summarized in table III.

We only present data for FACE, since our experiments with `HySAT` were not successful. `HySAT` v0.8.5β ended in an allocation failure even for $n = 1$ and $\delta = 2^{-2}$. On the same example `HySAT` v0.8.4 ran for over 6 hours but did not report a conclusive answer in that given time-frame. Furthermore, in these experiments, we manually changed the constants in the verification model for FACE. We used a rational number representation for δ which caused significant performance improvements over table II. Hence, we used only a 10 minute time-out setting. In addition to the length n of the array with unknown positive integral content, we also provide the depth d of the expected witness trace.

The chosen values for δ range from 2^{-22} to 2^{-2}. Since FACE uses infinite precision reasoning, one may expect certain properties to be *easier* to solve for larger values of δ. On the other hand, a too large value for δ can cause spurious counterexamples, as is evidenced in table III in the case for $\delta = 2^{-8}$ when $n \geq 10$. A spurious entry for a fixed δ implies that we should find spurious entries for larger n modulo verification time-outs.

Moreover, note that in certain instances the depth of the expected witness trace is not the main decisive factor to predict whether a particular problem can be solved for a fixed δ. Consider, for example, the case for $\delta = 2^{-9}$ and $n \in \{20, 25, 30\}$. For $n = 20$, FACE provides the expected result, whereas for $n = 30$ it provides a spurious witness. We observe a time-out for $n = 25$, presumably due to the fact that the correct answer requires very fine precision. Similarly, for a fixed analysis depth, the analysis time is not predictable in terms of a rounding precision, as can be seen for $n = 25$ and $\delta \in [2^{-12}, 2^{-6}]$.

V. CONCLUSIONS AND FUTURE WORK

This paper presented a technique to analyze source code for numerical stability properties. We target numerical programs implemented using the IEEE 754 floating-point standard, and the precision loss incurred in such programs. There have been techniques that handle analysis of such numerical programs using abstract interpretation based techniques. We use bounded model checking (BMC) based on Satisfiability Modulo Theory (SMT) solvers to analyze programs with floating-point operations. We generate a mixed integral-real model that is then analyzed by a set of backend model checkers.

Based on our prior experience with bounded model checking techniques using SMT solvers, we have encoded a number of design choices into a static model. However, we believe that many of these design choices can be relaxed and used in an on-demand fashion. In the future, we envision the models to be dynamically refined in a counterexample-guided abstraction-refinement (CEGAR) fashion [7]. Furthermore, our experiments on some benchmarks have indicated that further improvements to the model checkers based on SMT solvers

$\log_2 \delta$	n=5(d=68)	n=10(d=118)	n=15(d=168)	n=20(d=218)	n=25(d=268)	n=30(d=318)	n=35(d=368)
-22	2.2s	23.3s	440.8s	time-out	60.5s	279.7s	time-out
-20	10.1s	15.1s	36.8s	time-out	93.5s	571.8s	time-out
-18	2.3s	15.6s	time-out	123.5s	185.3s	197.3s	time-out
-16	2.7s	5.1s	8.2s	162.9s	32.9s	149.1s	time-out
-14	2.5s	11.6s	35.0s	34.9s	577.2s	time-out	time-out
-12	1.1s	7.8s	58.0s	16.4s	91.3s	175.4s	time-out
-10	1.2s	6.8s	34.3s	24.3s	162.8s	245.5s	time-out
-9	1.3s	36.5s	34.5s	61.7s	time-out	spurious(273.4s)	spurious(457.8s)
-8	0.8s	spurious(14.1s)	spurious(52.6s)	spurious(56.8s)	spurious(120.5s)	spurious(226.3s)	time-out
-6	spurious(0.9s)	spurious(2.8s)	spurious(8.9s)	spurious(43.6s)	spurious(97.9s)	spurious(231.7s)	time-out
-4	spurious(0.8s)	spurious(3.4s)	spurious(32.2s)	spurious(73.5s)	spurious(79.5s)	spurious(229.6s)	spurious(496.5s)
-2	spurious(0.9s)	spurious(2.4s)	spurious(34.5s)	spurious(51.5s)	spurious(78.0s)	spurious(44.3s)	spurious(435.4s)

TABLE III
EFFECT OF CHANGING ROUNDING PRECISION ON MODEL CHECKER PERFORMANCE

is required to scale to larger pieces of code. For numerical stability problems, as mentioned before, the relevant size of programs is often limited since numerically unstable computations can often be found in small portions of a program.

Finally, we believe that a tighter integration between an abstract interpretation based tool such as FLUCTUAT with the technique presented here would benefit the user tremendously. In this paper we utilized user-provided invariants only to simplify the model. However, the warnings produced by FLUCTUAT or similar tools can now be concretized to generate a) concrete traces and b) external input values that show a particular numerical instability occur in practice.

ACKNOWLEDGEMENTS

We thank Christian Herde from the HySAT team for prompt responses to our many queries on the input language of HySAT as well as providing bug fixes.

REFERENCES

[1] M.V.A. Andrade, J.L.D. Comba, and J. Stolfi. Affine arithmetic. In *INTERVAL'94*, March 1994.
[2] R. Andraka. A survey of CORDIC algorithms for FPGA based computers. In *ACM/SIGDA sixth international symposium on Field programmable gate arrays (FPGA)*, pages 191–200, New York, NY, USA, 1998. ACM.
[3] A. Armando, J. Mantovan, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. J. Softw. Tools Technol. Transf. (STTT)*, 11(1):69–83, 2009.
[4] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, O. Wei, and A. Gupta. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *SAS*, pages 238–254. Springer, 2008.
[5] T. Ball and S. Rajamani. The SLAM toolkit. In *Computer Aided Verification, 13th International Conference*, 2001.
[6] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in Systems Design*, 19(1):7–34, 2001.
[7] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
[8] E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2004. To appear.
[9] E.M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. In *Model Checking for Dependable Software-Intensive Systems Workshop*, 2003.
[10] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In T. Ball and R.B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 415–418. Springer, 2006.
[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In S. Sagiv, editor, *14th European Symposium on Programming (ESOP)*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, (5):394–397, 1962.
[13] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, (7):7–201, 1960.
[14] B. Dutertre and L.M. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R.B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
[15] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1(3-4):209–236, 2007.
[16] M.K. Ganai. Efficient decision procedure for bounded integer non-linear operations using SMT(LIA). In *Haifa Verification Conference (HVC)*, volume 5394 of *LNCS*, pages 68–83. Springer, 2008.
[17] M.K. Ganai, A. Gupta, F. Ivančić, V. Kahlon, W. Li, N. Papakonstantinou, S. Sankaranarayanan, and C. Wang. Towards precise and scalable verification of embedded software. In *Design and Verification Conference (DVCon)*, February 2008.
[18] M.K. Ganai and F. Ivančić. Efficient decision procedure for non-linear arithmetic constraints using CORDIC, 2009. Under submission.
[19] E. Goubault. Static analyses of the precision of floating-point operations. In P. Cousot, editor, *Symposium on Static Analysis (SAS)*, volume 2126 of *LNCS*, pages 234–259. Springer, 2001.
[20] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In S. Leue and P. Merino, editors, *Formal Methods for Industrial Critical Systems (FMICS)*, volume 4916 of *LNCS*, pages 3–20. Springer, 2007.
[21] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), April 2000.
[22] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.
[23] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, June 2001.
[24] IEEE. 754-2008: IEEE standard for floating-point arithmetic, August 2008. http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.
[25] F. Ivančić, I. Shlyakhter, A. Gupta, M.K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-SOFT. In *ICCD*, pages 297–308. IEEE Computer Society, 2005.
[26] V. Kahlon, A. Gupta, and N. Sinha. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *CAV*, pages 286–299. Springer, 2006.
[27] J. Krinke. Context-sensitive slicing of concurrent programs. In *9th European software engineering conference*, pages 178–187. ACM Press, 2003.
[28] MathWorks. PolySpace program analysis tool. http://www.polyspace.com.
[29] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.
[30] S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis using symbolic ranges. In *SAS*, pages 366–383. Springer, 2007.
[31] L. Séméria and G. de Micheli. SpC: Synthesis of pointers in C, application of pointer analysis to the behavioral synthesis from C. In *ICCAD*, pages 321–326. IEEE/ACM, November 1998.
[32] J.E. Volder. The CORDIC trigonometric computing technique. *IRE Trans. on Electronic Computers*, EC-8(3):330–334, 1959.
[33] J.S. Walther. A unified algorithm for elementary functions. In *AFIPS Spring Joint Computer Conference*, 1975.