Numerical Stability Analysis of Floating-Point Computations using Software Model Checking

Franjo Ivančić*, Malay K. Ganai*, Sriram Sankaranarayanan† and Aarti Gupta*
* *NEC Laboratories America, Princeton, NJ*
† *University of Colorado at Boulder, Boulder, CO*

***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. We target verification of numerical programs that use floating-point types, to detect loss of numerical precision incurred in such programs. Techniques based on abstract interpretation have been used in the past for such analysis. We use bounded model checking (BMC) based on Satisfiability Modulo Theory (SMT) solvers, which work on a mixed integer-real model that we generate for programs with floating points. We have implemented these techniques in our software verification platform. We report experimental results on benchmark examples to study the effectiveness of model checking on such problems, and the effect of various model simplifications on the performance of model checking.**

I. Introduction

Due to the ubiquitous availability of real-time and cyber-systems that interact with physical environments, there is a great need to develop verification technologies that target the whole system. Analyzing software for its correctness is a key step in guaranteeing safety of many important real-time devices, such as medical devices, automobiles or airplanes.

Recently, there has been extensive research on model checking software programs, including [4], [10], [20], [21] and many others. These techniques cover many language features, but the focus has often been on memory correctness issues, without much emphasis on numeric computations. CBMC [9] accurately models floating-point operations, by generating a bit-blasted formula that is directly handled by a SAT solver in the backend. Recently, the floating-point handling in CBMC was improved using counterexample-guided mixed over- and under-approximations when analyzing the bit-blasted formula [6]. However, to the best of our knowledge, neither CBMC nor any other model checker directly models the variability in the value of floating-points that can result from precision loss incurred in floating-point representations and operations in a program. In practice, the precision loss due to numerically unstable implementations in hand-written or model-based automatically generated source code are often overlooked by control engineers that design the higher level control models. Thus, we need techniques to find such instabilities in source code.

On the other hand, several techniques based on abstract interpretation have targeted this need in the ever-growing embedded software domain. These techniques 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 techniques are available in tools such as `ASTRÉE` [5], [11], `FLUCTUAT` [19], and `PolySpace` [24]. They provide scalable analysis based on abstract interpretation by limiting precision in certain cases, for example due to widening of loops. However, such path-insensitivity often causes false warnings. Furthermore, abstract interpretation based tools generally do not generate concrete counterexamples. This diminishes the tool capability in helping the user distinguish between accurate warnings and spurious ones.

A. Motivating Example

We introduce a small instructive example, adapted from [18], 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 [18], 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.

Algorithm 1 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.

```
procedure CtrlTest(floats a1, a2, b1, b2, c1, c2)
    float x = (c1*b2-c2*b1)/(a1*b2-a2*b1);
    if  x ≥ 0.0f then return TRUE;
    else  return FALSE;
    end if
end procedure
```

The problem observed in this example is known to be due to *cancelation* effects. Cancelation occurs when very small numbers are computed by addition or subtraction of two large numbers. This leads to a large loss of precision in the so computed small number. In the above example, note that the sign of the floating-point variable `x` 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 since it models the bit-precise floating-point values, but not the variability in these values due to loss in numerical precision. Furthermore, the provided answer relies on a particular rounding mode. For other instructive examples showcasing numerically stable and unstable algorithms, the reader is referred to [18].

B. Modeling Floating-points and Numeric Stability

In this paper, we seek to detect program paths that lose significant precision in the floating-point computations. To this end, we introduce a mixed real-integer model for floating-point variables and operations involving floating-points in a program. Floating-point variables in the source code are associated with auxiliary variables in the model to track the variability in their values along computations. These auxiliary variables contain variables in the reals, as well as variables of a small enumerated type denoting whether the floating-point variable is `NaN` (*not a number*), some infinity denoted as `Inf`, or an actual number. The auxiliary variables that are of type real represent a bounding interval in the model for the associated floating-point variable, as long as the type of the variable is *numeric*, i.e., it is not `NaN` or some `Inf`. We use this bounding interval to compute numeric errors.

For each floating point type (e.g. `float, double, long double` in C), we introduce a parametric constant δ that can be varied to achieve the desired level of precision in modeling *relative* numeric errors. By keeping δ close to the maximal relative error for that type, we can avoid too many false warnings, although this may make analysis more expensive. By increasing δ, we can keep the model sound, but this may introduce spurious warnings due to an overestimation of the relative rounding error. Similarly, we introduce a parametric constant ϵ that can be varied to achieve the desired level of precision in modeling *absolute* numeric errors in the *sub-normal* number range.

We target detection of the following kinds of numeric instabilities:

- Non-numeric type error: When an operation results in a *non-numeric* type.
- Large numeric error: When the potential error in a floating-point variable is larger than some (user provided) relative or absolute threshold.

In addition, we can handle arbitrary assertions that utilize modeling/auxiliary variables described above. In practice, we provide pre-designed macros to facilitate assertion writing. When our technique reports a numerically unstable path, the user can utilize this information to improve the stability of the algorithm. Finally, it should be noted that the modeling presented here can also be utilized to find other bugs in the program, including buffer overflows, that may depend on floating-point computations.

The main considerations in designing our model are to allow efficient use of both abstract interpretation and SMT-based bounded model checking (BMC) techniques. We can use abstract interpretation to find easy proofs (i.e. no errors possible) and to simplify the models before performing BMC, which is used to find concrete computation paths violating numeric stability. We would like to note that in typical embedded software, the numeric algorithmic core computation is often relatively short, as observed by the users and developers of FLUCTUAT. This indicates that model checking is potentially feasible, despite the increase in size of models due to auxiliary variables for floating-points and their updates (described in detail later).

C. Paper Contributions and Outline

This paper presents a software verification approach for finding numerically unstable computations in source code. We present a model that directly tracks the variability in floating-point values and provides complete handling of floating-points in the C language. Specifically, we describe modeling of *arithmetic operations*, *casting* between floating-point types and other types, as well as modeling of arbitrary *rounding* in floating-point computations and *sub-normal* numbers, as defined in the IEEE 754 floating-point standard.

We target detection of a large class of numeric stability problems – non-numeric values, and large relative/absolute errors. We use bounded model checking to find concrete error traces. We describe various modeling and analysis decisions that enhance the scalability of BMC.

We have implemented our model generation and model checking in the F-SOFT software verification platform [23] and present various experimental results. These results show that BMC is successful in finding concrete error traces with numeric instability in C programs. In an extended set of experiments, we also show that the performance of BMC can be significantly improved by providing a priori computed invariants to simplify the models. Although we have not yet fully implemented the abstract interpretation analysis to generate the invariants, our BMC results show that such analysis can be very valuable in helping BMC scale to larger programs. Finally, we study the effects of varying the parametric constants (δ, ϵ) that control the level of precision in our models and analyses.

The next section provides background on the IEEE 754 floating-point standard, and existing abstract interpretation techniques for handling floating-points. Section III describes our modeling of floating-point variables, floating-point operations, casting operations, and rounding. Section IV describes our implementation of model generation and BMC in the F-SOFT platform. Section V describes the experimental results. Finally, section VI concludes the paper with some remarks and an overview of further research directions.

II. BACKGROUND

A. The IEEE 754 Floating-Point Standard

We analyze source code with respect to the binary formats of the new IEEE 754-2008 standard [22], 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 144 is stored 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).[1]
- 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`).

1) Floating-point formats: For ease of presentation of the IEEE standard in this section, we focus on two floating-point formats defined in the standard: single-precision (specified using the keyword `float` in C/C++) and double-precision (`double`). Single-precision defines that a number be 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 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 $1.8 \cdot 10^{308}$.

2) Rounding: The IEEE754-2008 standard defines five rounding modes for each floating-point operation. There are two modes that *round to nearest* neighboring floating-point, where a bias can be set to *even numbers* or *away from zero* when the operation lands exactly midway between two representable floating-points. The other rounding modes are *rounding towards zero*, *rounding towards* ∞ and *rounding towards* $-\infty$. Every arithmetic operation is calculated precisely before rounding. There is a maximal relative error due to rounding for values resulting in the normalized number range. The maximal relative error for normalized numbers is 2^{-23} in single-precision and 2^{-52} for double-precision.

3) Sub-normal numbers: To provide *gradual underflow* near zero, 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 is that it guarantees that two nearby but different floating-point numbers always have a non-zero distance. However, note that operations that result in numbers in the sub-normal number range can have very large relative errors.

4) Operations: The standard defines the expected precision, expected results and exception handling for a variety of operations such as arithmetic operations (add, multiply, divide, square root, ...), casting conversions (between formats, to and from integral types, ...), comparisons, classification and testing for `NaN`, and many more. In this paper, we focus on arithmetic operations and casting operations, using the rounding precision prescribed by the standard.

B. Abstract Interpretation Techniques for Floating-points

There have been a number of proposed solutions for 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*.

Interval arithmetic. In order to quantify rounding errors in mathematical computations and to guarantee reliable results in numerical methods, mathematicians have studied interval arithmetic (IA) for many decades [25]. Instead of defining arithmetic operations on individual numbers, IA defines arithmetic operations on intervals of 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 operations, consider the addition or multiplication of two intervals $[\underline{x}, \overline{x}], [\underline{y}, \overline{y}]$ which are defined as $[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$, and $[\underline{x}, \overline{x}] \cdot [\underline{y}, \overline{y}] = [\min(\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y}), \max(\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y})]$, respectively. In a non-disjunctive domain analysis

[1]The standard treats ± 0 as two separate floating-point numbers since certain operations yield different results based on the *sign of zero*. For example, $1/(+0) =$ `Inf`, whereas $1/(-0) = -$`Inf`.

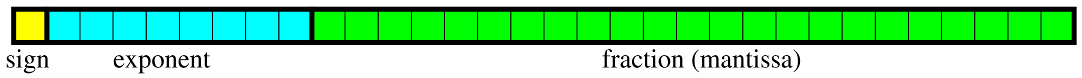

Figure 1. General layout of a floating-point number

setting, division by an interval that contains zero is defined to result in the interval $[-\infty, \infty]$.

Affine arithmetic. Affine arithmetic (AA) is an improvement over IA by tracking dependencies between variables, thus often providing more precise results than IA [1]. 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 (called weights), 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. These errors may come from input data uncertainty, formula truncation, or rounding. An error variable that appears in two quantities x and y implies that their errors 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 $5+w_0 \cdot \epsilon_2$, where w_0, ϵ_2 are introduced to model arithmetic rounding, and w_0 is a very small quantity representing machine precision.

The abstract interpretation based tool FLUCTUAT [19] 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 $x_0, x_1, \ldots, x_k$ using widening techniques inside loops in the program. It has been successfully used on small, but numerically important, parts of embedded controllers and discovered numerically unstable computations.

III. Modeling Floating-Points in Programs

Our modeling approach is largely inspired by FLUCTUAT. However, we use an IA-based approach instead of an AA-based approach, for reasons of scalability during model checking. Consider the number of state variables in the two modeling paradigms. In an IA-based model, we require $2|V_F|$ real variables to represent intervals for a set of floating-point variables V_F. Operations are handled as updates to these variables. However, a model using AA requires $|F|$ variables to model the weight of *each operation* in the set of floating-point operations F. This can be simplified by providing a heuristic deciding which weights and error terms to join together. However, such heuristics are very hard to generalize for arbitrary source code. Furthermore, our experience in a different setting [15] (using AA to check robustness of Simulink simulations) suggests that AA has scaling issues in symbolic execution-based approaches.

Hence, we pursue an IA-based approach for model checking purposes. One main reason in abstract interpretation techniques to move from IA to AA is to provide additional analysis precision. Rather than looking for more precise domains, we would like to explore whether the increased path-sensitivity in model checking is successful in detecting numerical instabilities in practice. Additionally, we expect savings by combining abstract interpretation with model checking, as we have done for other properties in F-Soft.

A. Auxiliary variables for floating-points

We introduce a number of auxiliary 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 ±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 floating-point status flags. The variable $\check{\mathtt{f}}$ represents the status of the lower interval bound variable $\underline{\mathtt{f}}$, and $\hat{\mathtt{f}}$ represents the status of the upper interval bound variable $\overline{\mathtt{f}}$. Note that $\mathcal{F}$ does not distinguish between zero, normalized or sub-normal 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: $\hat{\mathtt{f}} = \mathtt{Number} \rightarrow \mathtt{f} \leq \overline{\mathtt{f}}$.

B. Handling arithmetic operations

For every arithmetic floating-point operation, such as z := x$+_F$y, we add additional statements to the model that update the related auxiliary 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 ∞. 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 functions $\downarrow, \uparrow$ depend 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 assignment to z, we introduce the following 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 $\hat{\mathtt{z}}$. Note that these updates to the bound variables and the status flags are based on the semantics of the IEEE standard. For example, the standard prescribes that $\mathtt{Inf}+(-\mathtt{Inf})=\mathtt{NaN}$.

While the update to the floating-point status variables can be quite complex, as shown above, note that these variables only range over $\mathcal{F}$. Often, these updates can be simplified by a priori computed invariants in the program, thus allowing us to resolve many of the conditionals before generating this complex update function.

Finally, note that we change the type of the original floating-point variable in the program to be of type real in the model. Thus, each variable in the model is either an integer or of real type.

C. Handling Casting Operations

In addition to handling arithmetic operations, we also model casting operations between floating-point types and between floating-point variables and integer type variables. Such casts can frequently lead to loss of precision.

We support the following casting operations:

- 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. It may also cause a change in the floating-point status from `Number` to $\pm$`Inf`.
- Casting from an integer to a floating-point variable requires rounding.
- Casting from floating-point to an integer variable is only well defined if the integer variable is large enough to represent the integral part of the floating-point value.

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, and later on the sub-normalized number range.

Rounding normalized numbers. The maximal relative precision for normalized numbers is fixed for each floating-point type. For a floating-point type $t \in \mathcal{T}$, we introduce a parametric constant δ_t that is larger than or equal to the worst-case relative error in the normalized number range for t. For ease of presentation, we simply write δ instead of δ_t.

We can vary δ in our model, to tradeoff between the precision and performance of our analysis. By keeping δ close to the maximal relative error we can avoid too many false warnings, while it may make the analysis stage more expensive. By increasing δ we keep the analysis model sound, but may introduce spurious warnings due to an overestimation of the relative rounding error.

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

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

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

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. Instead, we use absolute error modeling in the sub-normal number range.

We choose a parametric constant ϵ_t for $t \in \mathcal{T}$ that is larger than or equal to the largest absolute error in the sub-normal number range of t. We use absolute error based modeling for operations resulting in the sub-normal number range. Thus, we define rounding functions $\uparrow_s, \downarrow_s \colon \mathbb{R} \to \mathbb{R}$ for the sub-normal number range as: $\downarrow_s(x) := x - \epsilon$, and $\uparrow_s(x) := x + \epsilon$. The main disadvantage of this model is that many operations that 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 domain analysis setting.

To avoid the spilling of interval ranges to straddle zero, we could also define a rounding function that preserves the sign of the bounding variable. Finally, it is also possible to split the sub-normal number range into additional regions which may provide better accuracy. Each choice has its advantage and disadvantages where precision is traded-off against simplicity of the generated model.

Combining rounding functions. A straightforward solution for 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. However, this would introduce additional floating-point type specific constants delineating the boundary between normalized numbers and sub-normal numbers. Furthermore, each expression would require an additional ITE (*if-then-else*) to denote this choice.

In order to simplify the expressions for analysis, we define $\uparrow, \downarrow \colon \mathbb{R} \to \mathbb{R}$ to be $\downarrow := \downarrow_n + \downarrow_s$, and $\uparrow := \uparrow_n + \uparrow_s$. Note that this modeling is sound since we always enforce both error types although only one is applicable at a time.

Furthermore, note that the absolute error ϵ introduced in $\uparrow_s, \downarrow_s$ is very small and is immaterial as long as numbers are not close to the sub-normal range.

IV. Model Generation and Model Checking: Implementation

We have implemented our model generation for programs with floating-points, and bounded model checking for detecting numerical instability errors in the F-Soft platform [23].

In general, F-Soft provides a unified infrastructure to build a model that is utilized by both abstract interpretation techniques [3], and model checking techniques. Source code written in C/C++ is analyzed for user-specified properties, user-provided interface contracts, or standard runtime bugs such as buffer overflows, pointer validity, string operations, etc. Abstract interpretation in F-Soft is used to eliminate easy proofs. It is also used to perform simplifications on the model to reduce its size before model checking. F-Soft supports unbounded model checking as well as bounded model checking (BMC), where BMC can utilize either a SAT-solver [7] or an SMT-solver [2].

The verification model generated by F-Soft is a control-flow graph representation over integer variables and variables in the real domain. These models are analyzed in the back-end by model checkers that reason about non-linear mixed integer and real constraints. For this paper, we used Face [17], which performs bounded model checking using SMT-solvers and can prove some model properties in certain instances. HySAT [16] is a related model checker, but it was not able to successfully analyze the models generated here.

HySAT. HySAT is a satisfiability checker for Boolean combinations of arithmetic constraints over reals and integers 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 [16].

Face. Face [17] is our in-house SMT-based model checker for models with Boolean combinations of arithmetic constraints over real- and integer-valued variables. Face utilizes CORDIC algorithms [27], [28] to translate non-linear arithmetic into linear arithmetic given some precision requirement. These algorithms are 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) (Satisfiability Modulo Theory for Linear Real Arithmetic) solver such as Yices [14] 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.

V. Experiments

A. Motivating Example Revisited

This section presents the automatically generated model for the motivating example introduced in Section I. After some automatic rewriting (utilizing temporary variables for individual floating-point operations, and removal of constants $c_1 = 0.0$ and $c_2 = 1.0$ and associated computations that do not require rounding), the computation is split into the following sequence:

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

We analyze each computation step for stability in the following sense: it should result in an interval with numeric bounds, and it should produce a small interval. In the following experiments, we defined this to have a small relative error, specified here to be 0.1%. That is, after each operation we checked the result using the method `checkStability(x)` defined here assuming $\min(|\underline{x}|, |\overline{x}|) \neq 0$ as:

$$\text{checkStability}(x) \;:=\; \check{\mathrm{x}} = \texttt{Number} \wedge \hat{\mathrm{x}} = \texttt{Number} \wedge \frac{\overline{x} - \underline{x}}{\min(|\underline{x}|, |\overline{x}|)} < 10^{-3}.$$

We present the automatically generated model for the first three steps of the simplified program shown above in Algorithm 2. Note the use of the pre-defined macro `checkStability(x)` to add assertions that are checked by the model checker. To simplify the presentation, we avoid showing the last step involving the division, as well as the definitions for the updates to the floating-point status variables for multiplication operations $\mathcal{L}_M$ and $\mathcal{U}_M$ and for subtraction operations $\mathcal{L}_S$ and $\mathcal{U}_S$. Note that the updates for subtraction are similar to those presented for addition in Section III-B. The definition of the updates for multiplication are much more involved. As we mentioned earlier, a priori computed invariants can simplify these updates considerably.

The computations of `a1b2` and `a2b1` are deemed to be stable by our analysis. 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 $6.0s$ (line 26). It shows a witness, where the lower bound of `denom` is negative, while the upper bound is positive. The witness

Algorithm 2 Simplified model for motivating example of Algorithm 1

```
1: procedure MODEL
2:     $\underline{a1} =\downarrow (37,639,840)$
3:     $\overline{a1} =\uparrow (37,639,840)$
4:     $\underline{a2} =\downarrow (29,180,479)$
5:     $\overline{a2} =\uparrow (29,180,479)$
6:     $\underline{b1} =\downarrow (-46,099,201)$
7:     $\overline{b1} =\uparrow (-46,099,201)$
8:     $\underline{b2} =\downarrow (-35,738,642)$
9:     $\overline{b2} =\uparrow (-35,738,642)$
10:    $\check{\texttt{a1}} = \hat{\texttt{a1}} = \check{\texttt{a2}} = \hat{\texttt{a2}} = \texttt{Number}$
11:    $\check{\texttt{b1}} = \hat{\texttt{b1}} = \check{\texttt{b2}} = \hat{\texttt{b2}} = \texttt{Number}$
12:    $\underline{a1b2} =\downarrow \min(\underline{a1}\cdot\underline{b2}, \underline{a1}\cdot\overline{b2}, \overline{a1}\cdot\underline{b2}, \overline{a1}\cdot\overline{b2})$
13:    $\overline{a1b2} =\uparrow \max(\underline{a1}\cdot\underline{b2}, \underline{a1}\cdot\overline{b2}, \overline{a1}\cdot\underline{b2}, \overline{a1}\cdot\overline{b2})$
14:    $\check{\texttt{a1b2}} = \mathcal{L}_M(\underline{a1}, \overline{a1}, \check{\texttt{a1}}, \hat{\texttt{a1}}, \underline{b2}, \overline{b2}, \check{\texttt{b2}}, \hat{\texttt{b2}})$
15:    $\hat{\texttt{a1b2}} = \mathcal{U}_M(\underline{a1}, \overline{a1}, \check{\texttt{a1}}, \hat{\texttt{a1}}, \underline{b2}, \overline{b2}, \check{\texttt{b2}}, \hat{\texttt{b2}})$
16:    checkStability$(a1b2)$
17:    $\underline{a2b1} =\downarrow \min(\underline{a2}\cdot\underline{b1}, \underline{a2}\cdot\overline{b1}, \overline{a2}\cdot\underline{b1}, \overline{a2}\cdot\overline{b1})$
18:    $\overline{a2b1} =\uparrow \max(\underline{a2}\cdot\underline{b1}, \underline{a2}\cdot\overline{b1}, \overline{a2}\cdot\underline{b1}, \overline{a2}\cdot\overline{b1})$
19:    $\check{\texttt{a2b1}} = \mathcal{L}_M(\underline{a2}, \overline{a2}, \check{\texttt{a2}}, \hat{\texttt{a2}}, \underline{b1}, \overline{b1}, \check{\texttt{b1}}, \hat{\texttt{b1}})$
20:    $\hat{\texttt{a2b1}} = \mathcal{U}_M(\underline{a2}, \overline{a2}, \check{\texttt{a2}}, \hat{\texttt{a2}}, \underline{b1}, \overline{b1}, \check{\texttt{b1}}, \hat{\texttt{b1}})$
21:    checkStability$(a2b1)$
22:    $\underline{denom} =\downarrow (\underline{a2b1} - \overline{a1b2})$
23:    $\overline{denom} =\uparrow (\overline{a2b1} - \underline{a1b2})$
24:    $\check{\texttt{denom}} = \mathcal{L}_S(\underline{a2b1}, \check{\texttt{a2b1}}, \overline{a1b2}, \hat{\texttt{a1b2}})$
25:    $\hat{\texttt{denom}} = \mathcal{U}_S(\overline{a2b1}, \hat{\texttt{a2b1}}, \underline{a1b2}, \check{\texttt{a1b2}})$
26:    checkStability$(denom)$
27: end procedure
```

produced by FACE interpreted as an interval may not be maximal. In this case, FACE produced the answer that `denom` may be within $[-4.8 \times 10^{13}, 5.0 \times 10^{-38}]$. Note that mathematically speaking the value of `denom` is -1.

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 in our assertion. 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 $[-8.0 \times 10^{12}, 2.2 \times 10^{10}]$, causing the following values for the status flags of `x`: $\check{\texttt{x}}$=`-Inf`, $\hat{\texttt{x}}$=`Inf`.

B. Effect of Simplifying Floating-point Status Flags

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. We compare the benefit of simplifying statements related to the floating-point status flags; i.e., a model where $\check{\texttt{f}}, \hat{\texttt{f}} \in \mathcal{F}$ are set to `Number`.

Consider the example shown in Algorithm 3. It computes the sum of an array with unknown data of type `int` and unknown array length. The model checker is asked to generate a large enough array and integer contents so that the sum of the array is bigger than a target floating-point number. By increasing the target number, we can generate bigger arrays and longer witness traces.

Algorithm 3 Integer array sum

```
Require: length(A) ≥ n
Require: target is numeric
1: procedure TESTSUM(int A[] , int n, float target)
2:     float sum = 0.0f
3:     for int i=0 to n-1 do
4:         float tmp = (float) A[i]
5:         sum += tmp
6:         assert(sum<target)
7:     end for
8: end procedure
```

The sum is computed using a loop that walks the array. Each loop iteration contains two rounding operations. First, we cast an integer to a floating-point. Second, we add two floating-points. 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 (see Section III-D for the definition of $\uparrow$ and $\downarrow$ utilizing the scalar multiplication in $\uparrow_n$ and $\downarrow_n$).

For each numeric target we create two models: The first model consists of the model as described in Section III. The second model propagates the fact that all floating-point variables are always of status `Number` in the model thus allowing constant folding to eliminate status flags from the resulting model. Note that these simplifications are valid in that this constraint is indeed an invariant of the system, for small enough `target` values resulting in small enough `n`. However, the model checker may have to discover this fact during its analysis.

In Figure 2, we present experimental results for this benchmark with increasing size, solved using FACE. Each data point corresponds to a case where the model checker is asked to find a progressively larger array between sizes 1 and 21. The x-axis shows the depth at which a witness is found in our models, ranging from depth 15 for the array of size 1, to depth 195 for the array of size 21. All experiments were run with a time-out of 1800 seconds on Linux servers. If a time-out occurred, we denote that in the figure with a value of 2000 seconds.

We compare the performance (cumulative runtime) of FACE on the complete model (denoted `Full model` and shown using blue crosses) against models where we simplified the status flags (denoted `Simplified` and shown using red squares). FACE was able to analyze the generated models up to depth 186 within the half hour time limit on the simplified model, whereas on the full model we could not find a witness beyond depth 168 within the time limit. In general, the model checker was able to find longer witnesses traces on the simplified models. However, in certain cases,

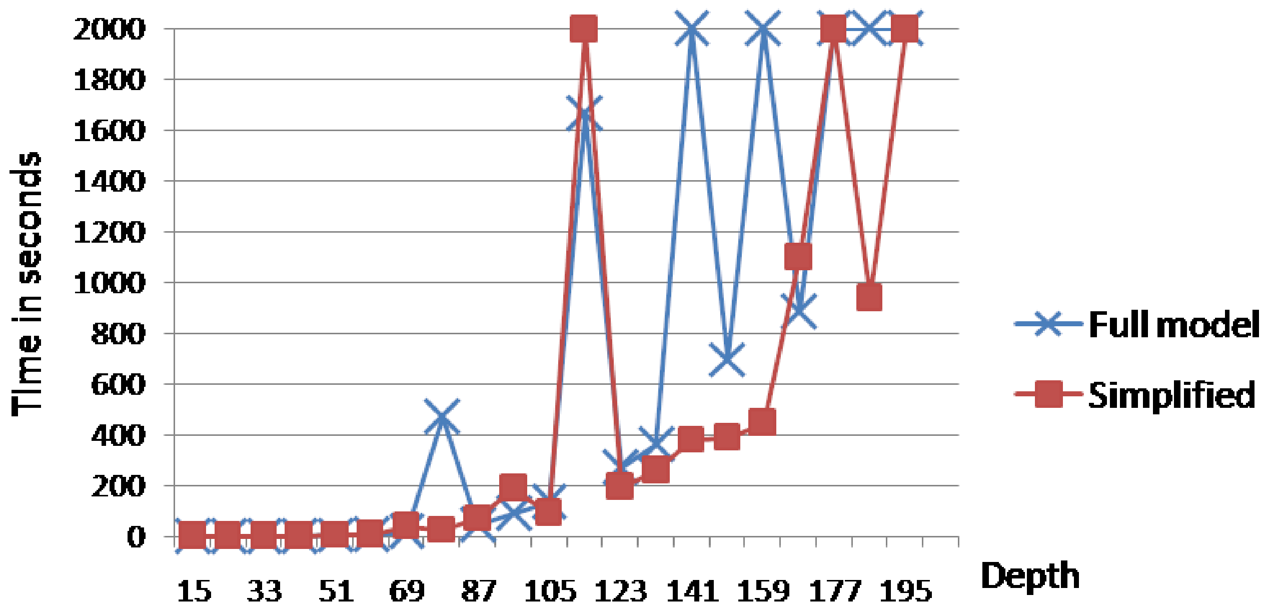

Figure 2. Effect of floating-point type simplification on model checking

such as for depth 114, the model checker finds a witness trace for the full model but not for the simplified model.

C. *Effect of Simplifying Floating-point Assignments*

In the previous section, we analyzed the benefit of an up-front simplification on the floating-point status flags. In this section, we further analyze the benefits that simplifying assignments to other auxiliary variables can provide. 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 target the following simplifications:

- We simplify assignments by predicting the sign of the result of a computation. This allows us to simplify $\downarrow, \uparrow$.
- We simplify assignments due to modeling of multiplication that contain $\min$ and $\max$ functions.
- We further simplify the assignments due to modeling of rounding by predicting whether the result of a computation is a sub-normal number.

We use a slightly altered program to generate a multitude of benchmarks. Again, the program computes the sum of an input array that contains unknown positive integer values. The sum is computed using floating point numbers, with a cast from `int` to `float`, as well as the rounding during each addition operation.

The property of interest here is that the maximal relative error of the computation is less than some threshold. We fixed the threshold to be 10% regardless of the length of the input array. Table I summarizes the results which were performed using FACE. `TO` denotes a time-out, and - means that we did not perform an experiment.

length	depth	full	simplified $\mathcal{F}$	simpler updates	range
1	28	1.2s	—	—	—
2	38	5.3s	—	—	—
3	48	TO	16.0s	—	—
4	58	—	90.9s	—	—
5	68	—	82.0s	—	—
6	78	—	TO	231.2s	106.5s
7	88	—	—	277.8s	57.9s
8	98	—	—	582.3s	TO
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	—	—	TO	2298.4s
14	158	—	—	TO	1636.5s
15	168	—	—	—	TO

Table I
EFFECT OF INVARIANT-BASED MODEL SIMPLIFICATIONS ON MODEL CHECKER PERFORMANCE

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 I. 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 cumulative BMC run-time of the complete model as discussed in this paper. The column `simplified` $\mathcal{F}$ provides the run-times for models where we utilized a priori invariants that each floating-point status variable is equal to `Number`.

The column `simpler updates` utilizes additional information about pre-computed invariants by simplifying the nested ITE structures. Finally, the column `range` shows the result, 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 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 significantly improve model checker performance by providing 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.

D. Effect of Varying Rounding Precision

In this section, we extend the example from Section V-C, and experiment with different values for the rounding precision δ, used in $\uparrow_n$ and $\downarrow_n$. The array is fixed to be of length $n \in [5, 35]$. All settings for δ provide sound answers to the verification problem at hand. The experiments are summarized in Table II.

In these experiments, we changed the representation of constants in the verification model from a scientific notation to a rational number representation for δ. This caused significant performance improvements using FACE over Table I, since LRA solvers such as Yices [14] handle rational numbers directly. Hence, we used only a 10 minute time-out setting. In addition to the length n of the array, we also note 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 (denoted `SP`), due to the overly imprecise modeling of rounding operations. This can be seen in Table II in the case for $\delta = 2^{-8}$ when $n \geq 10$. Note that a spurious entry for a fixed δ implies that we should find spurious entries for larger n (modulo verification time-outs).

In certain instances, the depth of the expected witness trace is not the decisive factor in predicting 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 $n = 25$, due to the fact that the search for the 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 $\delta \in [2^{-12}, 2^{-6}]$.

This set of experiments provides scope for further improvements, suggesting that counterexample-guided abstraction-refinement may guide the search for a suitable δ (and other parameters). Furthermore, we were intrigued by the substantial performance improvement of the model checker by changing the number representation format.

VI. Conclusions and Future Work

This paper presented a technique to detect numerical stability errors in source code. We target programs implemented using the IEEE 754 floating-point standard, and the precision loss incurred in such programs. Techniques based on abstract interpretation have been proposed in the past. We use bounded model checking based on Satisfiability Modulo Theory solvers to analyze programs with floating-point operations, where we automatically generate a mixed integral-real model that is analyzed by backend model checkers.

Based on our experience with bounded model checking techniques using SMT solvers, we have encoded a number of design choices into a static model. However, 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 [8]. Furthermore, our experiments on some benchmarks have indicated that further improvements to the model checkers based on SMT solvers is required to scale to larger models. The relevant size of programs for numerical instabilities is often limited since numerically unstable computations can often be found in small portions of a program. In addition to generating models for `HySAT` and FACE, we also generate non-linear SMT-LIB-compliant models. Note that we have made some related `HySAT` and SMT-LIB-compliant models available at [26].

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. In this paper we utilized user-provided invariants only to simplify the model. However, the warnings produced by `FLUCTUAT` or similar tools can also be used to generate a) concrete error traces and b) external input values which show that a particular numerical instability can 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` and for providing bug fixes.

References

[1] M.V.A. Andrade, J.L.D. Comba, and J. Stolfi. Affine arithmetic. In *INTERVAL'94*, March 1994.

[2] A. Armando, J. Mantovan, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *J. Softw. Tools Technol. Transf.*, 11(1):69–83, 2009.

$\log_2 \delta$	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	TO	60.5s	279.7s	TO
-20	10.1s	15.1s	36.8s	TO	93.5s	571.8s	TO
-18	2.3s	15.6s	TO	123.5s	185.3s	197.3s	TO
-16	2.7s	5.1s	8.2s	162.9s	32.9s	149.1s	TO
-14	2.5s	11.6s	35.0s	34.9s	577.2s	TO	TO
-12	1.1s	7.8s	58.0s	16.4s	91.3s	175.4s	TO
-10	1.2s	6.8s	34.3s	24.3s	162.8s	245.5s	TO
-9	1.3s	36.5s	34.5s	61.7s	TO	SP(273.4s)	SP(457.8s)
-8	0.8s	SP(14.1s)	SP(52.6s)	SP(56.8s)	SP(120.5s)	SP(226.3s)	TO
-6	SP(0.9s)	SP(2.8s)	SP(8.9s)	SP(43.6s)	SP(97.9s)	SP(231.7s)	TO
-4	SP(0.8s)	SP(3.4s)	SP(32.2s)	SP(73.5s)	SP(79.5s)	SP(229.6s)	SP(496.5s)
-2	SP(0.9s)	SP(2.4s)	SP(34.5s)	SP(51.5s)	SP(78.0s)	SP(44.3s)	SP(435.4s)

Table II
EFFECT OF VARYING ROUNDING PRECISION ON MODEL CHECKER PERFORMANCE, SP STANDS FOR SPURIOUS COUNTEREXAMPLE

[3] 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.

[4] T. Ball and S. Rajamani. The SLAM toolkit. In *CAV*, 2001.

[5] 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*. ACM, 2003.

[6] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*, 2009.

[7] 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.

[8] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[9] 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.

[10] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *Computer Aided Verification (CAV)*, 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 *ESOP*. 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 *Computer Aided Verification (CAV)*, pages 81–94. Springer, 2006.

[15] G.E. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Robustness of model-based simulations. In *30th IEEE Real-Time Systems Symposium*, 2009.

[16] 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.

[17] M.K. Ganai and F. Ivančić. Efficient decision procedure for non-linear arithmetic constraints using CORDIC. In *FMCAD*, November 2009.

[18] E. Goubault. Static analyses of the precision of floating-point operations. In *Symposium on Static Analysis (SAS)*, pages 234–259. Springer, 2001.

[19] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *FMICS*. Springer, 2007.

[20] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), April 2000.

[21] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[22] IEEE. 754-2008: IEEE standard for floating-point arithmetic, August 2008. http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

[23] 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*. IEEE Computer Society, 2005.

[24] MathWorks. PolySpace program analysis tool. http://www.polyspace.com.

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

[26] NEC Laboratories America. NECLA verification benchmarks. www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php.

[27] J.E. Volder. The CORDIC trigonometric computing technique. *IRE Trans. on Electronic Computers*, EC-8(3):330–334, 1959.

[28] J.S. Walther. A unified algorithm for elementary functions. In *AFIPS Spring Joint Computer Conference*, 1975.