Symbolic Execution for Checking the Accuracy of Floating-Point Programs [*]

Jaideep Ramachandran[†]
Northeastern University
jaideep@ccs.neu.edu

Corina Păsăreanu
NASA Ames Research Center/CMU
corina.s.pasareanu@nasa.gov

Thomas Wahl
Northeastern University
wahl@ccs.neu.edu

ABSTRACT

Programs with floating-point calculations tend to give rise to hard-to-predict behavior. Such uncertainty cannot be ignored: floating-point errors can have catastrophic consequences, as it happened with the Patriot missile accident in 1991. The likelihood of such incidents can be decreased by using automated technology to reliably analyze numerical code. We present a symbolic execution approach to checking the *accuracy* of numerical programs, investigating how much a floating-point computation deviates from the "ideal" computation on real values. Our method is implemented in the Symbolic PathFinder tool and leverages and extends the floating-point decision procedure REALIZER to check symbolic path constraints and to perform the accuracy checks. We further illustrate the possibility of using our tools to enhance abstract interpretation-based analyses to obtain tighter bounds on the numerical error introduced by floating-point computations. Initial experiments show the promise of our approach.

1. INTRODUCTION

Binary floating-point arithmetic (FPA) is the most widely used approximation of real arithmetic available on processors today. The necessity to round numerical results not only introduces inaccuracies, but often also renders arithmetic unintuitive, as witnessed by the loss of basic laws such as associativity of addition. As a result, program errors due to unfamiliarity with FPA are common, they are hard to track down, and they can be catastrophic: the Patriot missile accident in 1991 left 26 people dead after inadvertently hitting an Army installation [2]. The Ariane 5 rocket exploded during its first test flight in 1996 due to a floating-point conversion error, destroying 7 billion dollars worth of research and development.

[*]This research is supported in part by the US National Science Foundation under grant no. CCF-1218075, by Draper Laboratory and by SGT, Inc.

[†]Work was performed while doing an internship at NASA Ames Research Center.

In this paper we report on automated analysis techniques for floating point programs. Our techniques are based on symbolic execution, a popular method that executes programs on symbolic, rather than concrete, inputs. Symbolic execution results in a formula (path condition) such that an assignment to the program inputs satisfying the formula engenders a test case that forces control along the encoded path and thus covers it.

Symbolic execution tools have proved very successful at finding subtle errors in software. However, floating-point computations have not been modeled accurately, due to the lack (mostly) of background SMT solvers that can handle FPA. Instead, current symbolic tools typically resort to constraint solving using real arithmetic, which is insufficient for floating-point programs: code that is "correct" under real semantics may well be buggy under floating-point semantics, as the trivial example of loss of associativity mentioned above already illustrates.

Towards closing this gap, we have extended the Symbolic PathFinder (SPF) tool with capabilities for solving the constraints generated from flight software, which makes heavy use of floating point computations. The floating-point constraints generated by SPF are passed to REALIZER [7], a decision procedure for precise floating-point reasoning. The combination of SPF and REALIZER thus enables the accurate analysis of programs manipulating floating point numbers. Furthermore, as part of this work, we have extended REALIZER to precisely compute the *accuracy* of the program computations, i.e. to compute how much the floating-point computation deviates from the "ideal" computation (using real numbers). This involves a non-trivial extension of REALIZER with handling of a special kind of mixed floating-point and real constraints, with integer constraints used for specifying the desired numerical precision.

We have applied our tool chain successfully to checking whether a floating-point result is within a specified distance from the result of an ideal calculation over the reals. We have further investigated the possibility of using our tools to enhance abstract interpretation-based analyses to obtain tighter bounds on the numerical error introduced by floating-point computations. We are currently investigating inductive reasoning techniques for handling loops. Our preliminary experiments show the promise of our proposed techniques.

The paper is organized as follows. Section 2 provides background about the floating-point decision procedure REALIZER and the IEEE-754 2008 format for binary floating-point representation. In Section 3, we describe our extensions to REALIZER and present how we combined symbolic execution of SPF with the extended procedure to check accuracy of floating-point computation. We describe our experience with checking assertions in Java programs in Section 4 and provide preliminary results of how bounds obtained from a interval arithmetic-based tool can be improved for getting better accuracy bounds. In Section 5, we describe our work in progress, which is to extend accuracy checks of Section 3 to reasoning about computations in loops.

Related Work

We present a few pointers to most relevant related work. Abstract Interpretation (AI) tools like FLUCTUAT [4] and ASTREE use abstract numerical domains to reason about floating-point computations. Such reasoning tends to be efficient but "conservative", in that it often produces rather crude bounds for the deviation of the computation from real arithmetic. In contrast, we combine symbolic execution with an exact decision procedure to check exactly if the floating-point deviation from the ideal value is within the bound provided for the check.

Symbolic execution (e.g. in SPF) has been previously combined with interval solving and meta-heuristic search strategies to enhance numerical constraint solving. Although these strategies can solve constraints over complex mathematical functions, they reason about floating-point types in an inexact manner [9].

2. BACKGROUND

2.1 Symbolic Execution

Symbolic execution is a well-known program analysis technique that executes programs on unspecified inputs, by using symbolic instead of concrete input data. For each executed program path, the analysis builds a path condition PC, encoding the conditions characterizing the inputs that follow that path. This PC is checked for satisfiability using off-the-shelf solvers. If a PC becomes unsatisfiable then the corresponding path is not feasible and the analysis backtracks. Traditional applications of symbolic execution include test case generation and error detection, with many tools available [8]. Symbolic execution of looping programs may result in an infinite symbolic execution tree. For this reason, symbolic execution is typically run with a (user-specified) bound on the search depth.

In this work we use Symbolic Pathfinder (SPF) [8], a symbolic execution tool for Java bytecode. SPF has an interface to plug in different solvers, that handle different types of constraints, e.g. integers, strings, floating-point abstracted as reals, etc. However until now exact floating-point reasoning was unsupported. We have used this interface to plug in REALIZER for the work here.

2.2 Floating-point arithmetic

Floating-point is somewhat formalized in the 2008 binary IEEE-754 standard (referred to as the "Standard" in the sequel). Floating-point numbers are represented as bit-strings of three components, as shown in Figure 1 for single-precision and double-precision numbers (known as *floats* and *doubles* in C/Java-like programming languages). The semantics of this number representation is given by the numerical value of the floating-point number f represented in this format:

$$value(f) = (-1)^s \times 1.m \times 2^e \tag{1}$$

where $1.m$ stands for the rational number whose integral part is 1 and whose fractional part is the sequence of digits m.[1] Some bit-string patterns are reserved for certain special values, most notably *NaN* (denoting non-numerical computational results such as 0/0) and $\pm\infty$ (denoting *overflown* computational results, i.e. beyond the range permitted by the format).

Realizer

REALIZER is a floating-point decision procedure that translates formulas in FPA into *equivalent* formulas (i.e. without approximations) over the theory of real and integer arithmetic [7]. In conformance with the Standard, it does so by replacing floating-point expressions with "infinitely precise" real-arithmetic expressions, followed by *rounding* the result. More precisely, let $\diamond$ denote binary FPA operator, $\star \in \{+, -, *, /\}$ the corresponding real operator, and p be the floating-point precision. Then we have

$$x \diamond y = \frac{rd(\,(x \star y)\,/\,2^e \;\cdot\; 2^p)}{2^p} \cdot 2^e \tag{2}$$

where e is the (unbiased) exponent of the (precise) real result $x \star y$. The function rd depends on the rounding mode, but generally involves digit truncation. For example, for the IEEE *round-to-negative* mode, we have $rd = \lfloor\cdot\rfloor$, denoting the floor function, which returns the largest integer not larger than its real-valued argument. Rounding thus results in a combination of real and integer arithmetic (RIA). The translated expressions can then be reasoned about using an RIA-capable solver such as Z3 [3], which is used in REALIZER. An advantage of using REALIZER over other bit-precise solvers like MATHSAT [5] is that REALIZER can be extended to combine floating-point and real theories together.

3. CHECKING ACCURACY OF FLOATING-POINT COMPUTATIONS

To check programs manipulating floating point numbers we have incorporated REALIZER in the symbolic execution tool Symbolic PathFinder (SPF) to provide exact checks for the satisfiability of path conditions and to generate test values for floating point inputs.

Furthermore we designed and added a function *checkAccuracy(var,δ,type)* in the *Debug* class of SPF to check if under the given path condition, the floating-point value of the symbolic expression stored in *var* can deviate from its ideal real arithmetic value by more than a bound δ that is supplied externally: either by a user with domain knowledge or by an external analyzer like an abstract interpreter. As the underlying decision procedure, REALIZER, is exact, the result of this check is also exact. If the value can deviate, it

[1]According to the Standard, the exponent is actually *biased* (shifted) so that it ranges over non-negative integers. We omit this detail in this description. Also excluded are subnormal numbers and special values like *NaN*, $\pm\infty$.

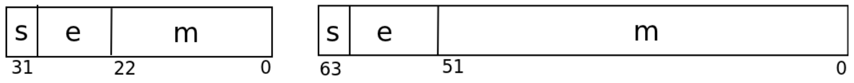

Figure 1: Single (32bit) and double precision (64bit) number representations (simplified). Components s, e, m are *sign bit*, *exponent*, *mantissa*, resp.

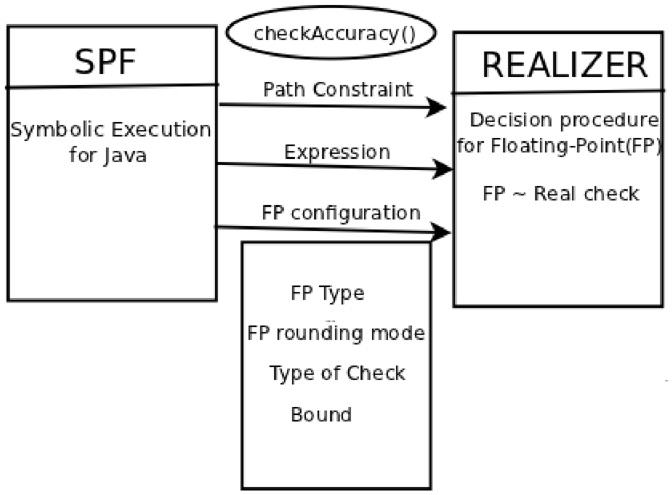

Figure 2: System architecture

provides an assignment to every input variable in the path condition or in the symbolic expression for the variable *var*. The parameter *type* can be absolute ("a") or relative ("r"), and is used to perform, respectively, an absolute or relative deviation check.

Adding this capability to Spf required an extension to Realizer, which is described next. A new variable to represent the Real arithmetic value of the symbolic variable being checked (*var* in this case) was defined and a constraint that assigns the symbolic expression for it, that is interpreted with Real semantics, was introduced. Another variable was introduced to represent the magnitude of the difference between the Real and the floating-point values, which is then compared with the absolute or relative bound, as required.

The set of additional constraints generated using the absolute accuracy-check mode added to Realizer is summarized in Figure 3. Here, *real-result* is the result obtained when operation *op* is performed on operands x and y, which are representable in the given Floating-Point format. *fp-result* denotes the rounded floating-point value obtained by rounding as described in Section 2. The final three constraints define the magnitude of the difference *diff* between *real-result* and *fp-result*. Note that there is a variable representing the real arithmetic result of every intermediate expression, and consequently also of the expression we want to check.

3.1 Integration in Symbolic PathFinder

We have used Spf's interface to plug in Realizer, an exact decision procedure for floating-point arithmetic, to reason about floating-point programs (figure 2). Whenever a new path condition is generated during symbolic execution, a query to check the satisfiability of that path is constructed and sent to the underlying constraint solver. Spf has a well-defined solver interface to enable addition of constraints to the path condition. In addition to the path condition to be checked for satisfiability, the expression to be checked for accuracy is passed on to Realizer. A configuration file is also written for Realizer to communicate parameters relevant in the context of floating-point computations. These include the floating-point type (e.g. float or double that determines the range and precision of numbers considered), the rounding mode (one of the 5 modes specified in IEEE-754), the type of accuracy check (absolute/relative) and the actual bound for the accuracy check. There exist other tools that can check floating-point assertions in programs, but to the best of our knowledge, this is the first instance where an exact check for accuracy is being performed using an exact floating-point reasoning engine.

3.2 Combining with Abstract Interpretation Tools

The *checkAccuracy()* function requires a bound to be specified as input, which is typically provided by a human expert. This may not be easy given the intricacies of floating-point arithmetic. AI tools have been used in the past to estimate deviation of floating-point computations from their ideal real arithmetic values, and this process is fully automated. We propose here to use an initial coarse bound obtained from an AI tool and tighten it further using *checkAccuracy()*. The motivation is to leverage the strengths of the respective techniques: precision of symbolic execution combined with a decision procedure, and efficiency of AI tools, to obtain tighter bounds on accuracies of numerical computations.

The initial over-approximate deviation value δ_0 obtained from an Abstract Interpretation tool can be further improved iteratively using *Debug.checkAccuracy()* to obtain a better bound for the deviation by using an iterative technique. We initialize δ, the deviation bound to be checked, with δ_0. At each iteration, *Debug.checkAccuracy()* checks if the floating-point value of the symbolic variable can deviate from its actual value by more than current δ. If this is the case, the obtained satisfying assignment is used for further inspection and analysis of the program. Otherwise, we decrease current δ further and use *Debug.checkAccuracy()* again. In practice, we can do a binary search in the interval [0,δ] until we have checked for the desired level of accuracy.

```
real-result = (x op y)              ; op = real-arithmetic operator
diff = (real-result - fp-result)    ; fp-result = rounded FP value
(diff < 0.0) => (abs-diff = (- diff))
(diff >= 0.0) => (abs-diff = diff)
(abs-diff > delta)
```

Figure 3: Absolute accuracy check: real vs. floating-point

```
void test(int N,float[] x){
 float sum = x[0];
 for (int i = 1; i < N; i++){
  sum = sum + x[i];
  assert(Debug.checkAccuracy(sum,0.01f,"a"));
 }
}
```

Figure 4: Regular summation

```
void test(int N,float[] x){
 float c=0.0f, sum=0.0f, y=0.0f, t=0.0f;
 y=x[0];  t=y; c=t-y; sum=t;
 for (int i = 1; i < N; i++){
  y = x[i] - c; t = sum + y;
  c = (t - sum) - y; sum = t;
  assert(Debug.checkAccuracy(sum,0.01f,"a"));
 }
}
```

Figure 5: Kahan summation

4. EXPERIENCE

We have evaluated our approach on several examples, described below.

4.1 Summation

Figure 4 shows an example program which computes the sum of an array of floating-point elements. Using SPF with *checkAccuracy()* check within the loop results in a violation after 2 additions, and an assignment $x[0] = -(8388607.0/128.0)$, $x[1] = -(8589935.0/8589934592.0)$, $x[2] = -(8388609.0/2147483648.0)$ was obtained when the range was set to between 10^{-3} and 10^5, and using *round-to-negative* mode. This also indicates that the value of *sum* could deviate further after more additions. Note that the check would not have failed over integers, over the reals or if the addition had been done using infinite-precision rational arithmetic.

Also, the counterexample is specific to this precision and rounding mode. In general, for a formula that is to be checked for satisfiability, the result (SAT/UNSAT) and the satisfying assignment, whenever one exists, is specific to the precision and rounding mode, that is, a satisfying assignment obtained for a given precision or rounding mode might not satisfy the same formula for a different precision or rounding mode.

We have also analyzed a modified summation algorithm, called Kahan summation [6], to perform a similar accuracy check. This summation algorithm, shown in Figure 5 was specifically designed to compensate the loss of accuracy for floating-point addition by using an extra variable that cancels out some of the loss across adjacent loop iterations. Using *checkAccuracy()* within the loop shows that indeed the intermediate and the final summation values are each within the deviation bound of 0.01 being checked (for $N = 15$).

Expression	GAPPA bound (w/o hint)	Improved bound	Improvement time(s)
x^3	0.0610962	0.00100	7
x^4	61.0963	0.06000	35
x^5	2000.0	0.00100	293

Table 1: Gappa bounds improved by our approach

4.2 Linear Controller

We have also analyzed a series of linear controllers (originally written in LUSTRE); Figure 6 shows the Java code corresponding to such a controller. Method *test* is invoked inside an infinite loop to mimic controller usage; `Verify.randomBool()` implements non-deterministic choice in SPF. The function *checkAccuracy*() was used to check that the floating-point deviation remained within the bound of 0.01 that we had specified for 2 iterations. With the symbolic expressions getting larger, certain calls to *checkAccuracy()* timed out (with timeout = 120s) for larger number of iterations, e.g. where the expression to be checked had more than 30 linear floating-point operators (including multiplications with constants that needed to be rounded). Our experiments with linear controllers showed the usefulness of our approach but also revealed scalability issues that we plan to address in future work.

4.3 Combining with Gappa

Table 1 shows the results of using REALIZER to improve upon the floating-point deviation bounds computed by GAPPA [1], a recent interval-based abstract interpretation tool, for some non-linear floating-point expressions. Column 1 gives the expression for which we are computing the deviation bound. The input variable x and all expressions are in the interval [-1000.0,1000.0]. Column 2 gives the bound computed by GAPPA without giving it any hints, and these results were obtained within a fraction of a second. Column 3 gives the bound obtained with the extension to REALIZER and Column 4 indicates the time taken for *checkAccuracy()* for that bound.

Although REALIZER needs to spend additional time to tighten the bound it has been provided, this can be useful, especially for critical software where this technique can be used to provide an improved assurance guarantee.

```
public void test(boolean r, float in0){
 if((in0 >= -1.0f) && (in0 <= 1.0f)){
  if(Verify.randomBool()){
    x0=1.0f; x1=1.0f; x2=1.0f; x3=1.0f; x4=1.0f;
   } else{
    x0=0.4250f * x0 + 0.8131f * in0;
    assert(Debug.checkAccuracy(x0,0.01f,"a"));
    x1=0.3167f * x0 + 0.1016f * x1 - 0.4444f *x2 + 0.1807f * in0;
    assert(Debug.checkAccuracy(x1,0.01f,"a"));
    x2=0.1278f * x0 + 0.4444f * x1 + 0.8207f  * x2 + 0.0729f * in0;
    assert(Debug.checkAccuracy(x2,0.01f,"a"));
    x3 =  0.0365f * x0 + 0.1270f * x1 + 0.5202f * x2 + 0.4163f * x3 - 0.5714f * x4 + 0.0208f * in0;
    assert(Debug.checkAccuracy(x3,0.01f,"a"));
    x4 =  0.0147f * x0 + 0.0512f * x1 + 0.2099f * x2 + 0.57104f * x3 + 0.7694f * x4 + 0.0084f * in0;
    assert(Debug.checkAccuracy(x4,0.01f,"a"));
   }}}
```

Figure 6: Linear Controller

5. HANDLING LOOPS

So far we have assumed that we analyze looping programs up to a given depth, as typical with symbolic execution approaches. Initial results are encouraging, and expected to scale to larger programs as our tool-chain matures. We are also working on reasoning about numerical deviation for computations within loops (e.g. that in Figure 6), by using the decision procedure to discharge *inductive* proof obligations constructed from symbolic execution as follows.

Let *expr* be the symbolic expression that is being modified within the loop. Let ${expr_i}^F$ and ${expr_i}^R$ denote the Floating-point and Real arithmetic symbolic expressions, respectively, of *expr* at the beginning of iteration i. Let δ_i be the symbolic expression representing the magnitude of the deviation of the ${expr_i}^F$ from ${expr_i}^R$ at the beginning of iteration i. We intend to prove the following using REALIZER

$$(|{expr_i}^R - {expr_i}^F| < \delta_i) \Rightarrow (|expr^R_{i+1} - expr^F_{i+1}| < \delta_{i+1})$$

Then we can bound the magnitude of the maximum deviation of $expr^F$ from $expr^R$ by $\lim_{i\to+\infty} \delta_i$ Also, in the above check, we can unroll loop body k times, for an appropriate k, instead of just once, like in k-induction.

The key challenge is to use an induction hypothesis that the underlying decision procedure is able to discharge. This requires coming with an appropriate value of δ_i, which captures the pattern of loss of accuracy just well enough for proving the bound.

6. CONCLUSION

We described a symbolic execution approach to checking the *accuracy* of numerical programs, investigating how much a floating-point computation deviates from the "ideal" computation on real values. Our method was implemented in the Symbolic PathFinder tool and leverages and extends the floating-point decision procedure REALIZER to check symbolic path constraints and to perform the accuracy checks. We also illustrated the combination of our proposed tool abstract interpretation-based analyses to obtain tighter bounds on the numerical error introduced by floating-point computations. Initial experiments showed the promise of our approach. In the future we plan to further investigate the handling of looping programs using invariants and to work on optimizing our implementation.

Acknowledgments

The authors would like to thank Pierre-Loic Garoche for providing the linear controller in Section 4.2, Arnaud Venet, Johann Schumann, Misty Davies, Timothy Wang and Peter Mehlitz for useful discussions and anonymous reviewers for helpful suggestions.

7. REFERENCES

[1] GAPPA. `http://gappa.gforge.inria.fr/`.

[2] Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. `http://klabs.org/richcontent/Reports/Failure_Reports/patriot/patriot_gao_145960.pdf`.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[4] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In *FMICS*, pages 53–69, 2009.

[5] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 131–140, 2012.

[6] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965.

[7] Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. Make It Real: Effective Floating-Point Reasoning via Exact Arithmetic. In *DATE*. IEEE, 2014.

[8] Corina S. Pasareanu and Neha Rungta. Symbolic Pathfinder: Symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[9] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Pasareanu. CORAL: solving complex constraints for symbolic pathfinder. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 359–374, 2011.