Mixed Abstractions for Floating-Point Arithmetic

Angelo Brillout
Computer Systems Institute, ETH Zurich

Daniel Kroening and Thomas Wahl
Oxford University Computing Laboratory

***Abstract*—Floating-point arithmetic is essential for many embedded and safety-critical systems, such as in the avionics industry. Inaccuracies in floating-point calculations can cause subtle changes of the control flow, potentially leading to disastrous errors. In this paper, we present a simple and general, yet powerful framework for building abstractions from formulas, and instantiate this framework to a bit-accurate, sound and complete decision procedure for IEEE-compliant binary floating-point arithmetic. Our procedure benefits in practice from its ability to flexibly harness both over- and underapproximations in the abstraction process. We demonstrate the potency of the procedure for the formal analysis of floating-point software.**

I. Introduction

Embedded systems are typically controlled by software that conceptually manipulates real-valued quantities, for instance measurements of environment data. Such quantities are stored in a computer as *floating-point numbers*. As only few real numbers can be encoded in this format, values must generally be *rounded* to some nearby floating-point number.

Compared to a computation with infinite precision, rounding can influence program behavior in multiple ways. The deviation caused by rounding can lead to unintuitive results, such as in a non-associative addition operation. Worse, the deviation can accumulate and eventually change the control flow of the program. Implementations of floating-point algorithms can be sensitive to very small variations in input. Bugs caused by such rounding errors are therefore often hard to reproduce and to test for, and have been referred to as "Heisenbugs" [1]. If undetected, they can have tragic consequences, as embedded devices are used in many mobile and ubiquitous computing environments. A prominent example is the Ariane 5 disaster, caused by an out-of-bounds 64-bit floating-point conversion. The indisputable need for reliability in embedded applications calls for precise and rigorous formal analysis methods.

Programs with floating-point arithmetic have been addressed in the past in various ways. In *abstract interpretation* [2], the program is (partially) executed on an abstract domain, such as real intervals. The generated transformations may, however, turn out too coarse for definite decisions on the given properties. *Proof assistants* are tools that prove theorems about programs (involving floating-point arithmetic) under human guidance. This guide, unfortunately, must be highly skilled to direct the tool towards a proof. Both abstract interpretation and theorem proving often lack the ability to generate counterexamples for invalid properties, which is essential for debugging and for the high-impact field of automated test-vector generation.

In this paper, we present a precise and sound decision procedure for (binary) floating-point arithmetic for the automatic analysis of software. A principal way of achieving this is to encode floating-point operations as functions on bit-vectors, and relying on efficient solvers for bit-vector logic, for instance those based on "bit-flattening" and subsequent SAT-solving. Unfortunately, this approach has proven to be intractable in practice, simply because it results in very large and hard-to-solve SAT instances, as we will illustrate.

A common solution to address this problem is to use *approximations* of formulas. Particularly, an overapproximation $\overline{\phi}$ simplifies the formula ϕ in a way that preserves all satisfying assignments; thus, unsatisfiability of $\overline{\phi}$ implies unsatisfiability of ϕ. Analogously, an underapproximation $\underline{\phi}$ simplifies ϕ in a way that removes satisfying assignments; thus, any satisfying assignment to $\underline{\phi}$ can be adjusted to one for ϕ.

Classical counterexample-guided abstraction refinement (*CEGAR*) [3] relies on overapproximations, which are refined if spurious counterexamples (in the form of spurious satisfying assignments) are encountered. In [4], a decision procedure for bit-vector arithmetic is presented that employs both types of approximations, although in a fixed alternation schedule. Such approaches are too rigid for floating-point arithmetic, as some formulas do not permit effective overapproximations, while others do not permit effective underapproximations.

In this paper, we propose a new abstraction method for checking the satisfiability of a floating-point formula ϕ. Our algorithm permits a mixed sequence S of both over- and underapproximating transformations. The formula ψ resulting from applying S to ϕ is in general neither an over- nor an underapproximation of ϕ. Our algorithm stops whenever (i) the simplified formula ψ permits a satisfying assignment that satisfies ϕ, too, or (ii) ψ is unsatisfiable and permits a resolution proof that is also a valid proof for ϕ. If neither of the opportunities (i) and (ii) applies, S needs to be refined. If ψ was found to be spuriously satisfiable, the algorithm removes an overapproximating transformation from S, otherwise an underapproximating transformation.

Our algorithm can be seen as a framework for a class of abstraction-based procedures to check the satisfiability of formulas in some logic: different choices of the transformation sequence S result in different instances of our framework. For example, the work presented in [4] is an instance where S strictly alternates between over- and underapproximations.

This research is supported by the the Toyota Motor Corporation, by the Semiconductor Research Corporation (SRC) under contract no. 2006-TJ-1539, by the EU FP7 STREP MOGENTES (project ID ICT-216679), and by the EPSRC project EP/G026254/1.

CEGAR is an instance where S contains only overapproximations. Our paper generalizes these methods in a way that permits a choice of approximations based on their effectiveness to simplify the input formula. Termination of any algorithm based on our framework is guaranteed as long as the sequence S can be shown to be depleted eventually.

We demonstrate the utility of the procedure on decision problems arising in bounded model checking (BMC) [5] of ANSI-C programs.

II. Preliminaries: Floating-Point Arithmetic

A. The IEEE floating-point format

The binary *floating-point format* is used to represent real numbers in a computer. Specifically, the triple consisting of a *sign* $s \in \{0,1\}$, an integer-valued *exponent* e, and a rational-valued *mantissa* m represents the floating-point number $(-1)^s \cdot m \cdot 2^e$. According to the IEEE standard 754, the three components are encoded using bit-vectors, resulting in the partitioned representation of a floating-point number shown in Figure 1.

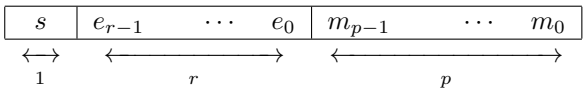

Fig. 1. The three fields of an IEEE-754 floating-point number

The sign bit s directly represents the sign of the floating-point number. The following two bit-vector fields are interpreted as follows:

- The bit field $\bar{e} := e_{r-1} \dots e_0$ encodes the integral exponent e as a binary number.
- Together with the hidden bit, the bit field $\bar{m} := m_{p-1} \dots m_0$ encodes the fractional value of the mantissa m; the representation ensures that $0 \le m < 2$. The hidden bit is derived from $\bar{e}$, and is used to distinguish normal and denormal numbers.

The widths r and p of the second and third bit fields in Figure 1 are called the *range* and the *precision* of the representation. The IEEE standard 754 defines two types of floating-point numbers: the *single* format with $(r,p) = (8,23)$, and the *double* format with $(r,p) = (11,52)$.

Unrepresentable real numbers are rounded, as we review in Section II-B. Numbers that are too large are represented using the symbols $-\infty$ and $+\infty$. The floating-point number NaN ("Not a Number") represents results of operations outside of real arithmetic, such as imaginary values. We call the floating-point numbers $\pm\infty$ and NaN *special*; they are represented using reserved patterns for exponent and mantissa.

In this paper, we manipulate floating-point formulas by varying the precision parameter p, while parameter r is fixed. We denote by $\mathbb{F}_p$ the set consisting of the floating-point numbers $(-1)^s \cdot m \cdot 2^e$ representable as a bit-vector with precision p, and the special numbers $\pm\infty$. With $\mathbb{R}^\infty := \mathbb{R} \cup \{\pm\infty\}$, we have $-\infty \le x \le +\infty$ for all $x \in \mathbb{R}^\infty$. Obviously, for $p' \le p$, we have $\mathbb{F}_{p'} \subseteq \mathbb{F}_p \subset \mathbb{R}^\infty$.

B. Floating-point arithmetic

The result of an operation $a \circ b$, for $\circ \in \{+,-,\times,/\}$, may not be representable in $\mathbb{F}_p$ even though a and b are in $\mathbb{F}_p$.[1] In such a case, an appropriate approximation is selected. For $x \in \mathbb{R}$, define the approximations $\lfloor x \rfloor_p$ and $\lceil x \rceil_p$ as

$$\begin{aligned} \lfloor x \rfloor_p &:= \max\{f \in \mathbb{F}_p : f \le x\}\,, \quad \text{and} \\ \lceil x \rceil_p &:= \min\{f \in \mathbb{F}_p : f \ge x\}\,. \end{aligned}$$

The values $\lfloor x \rfloor_p$ and $\lceil x \rceil_p$ are the two floating-point numbers in $\mathbb{F}_p$ nearest to x. There is no floating-point number strictly between $\lfloor x \rfloor_p$ and $\lceil x \rceil_p$. If x is larger than any non-special floating-point number in $\mathbb{F}_p$, then $\lceil x \rceil_p = +\infty$; analogously for $\lfloor x \rfloor_p$. The approximation values satisfy the following **nesting property**: for $p' \le p$, $\lfloor x \rfloor_{p'} \le \lfloor x \rfloor_p \le x \le \lceil x \rceil_p \le \lceil x \rceil_{p'}$.

Definition 1: A *rounding function* is a function $rd_p \colon \mathbb{R} \to \mathbb{F}_p$ such that, for all $x \in \mathbb{R}$, $rd_p(x) \in \{\lfloor x \rfloor_p, \lceil x \rceil_p\}$.

Specific rounding functions are also known as *rounding modes*. Two examples of rounding modes are *round-up* ($rd_p = \lceil \cdot \rceil_p$) and *round-down* ($rd_p = \lfloor \cdot \rfloor_p$).

The floating-point operators $\circledcirc \in \{\oplus, \ominus, \otimes, \oslash\}$ are defined as the *rounded* result of the corresponding real operators $\circ \in \{+,-,\times,/\}$:

Definition 2: For a given rounding function rd_p and an arithmetic operation $\circ \colon \mathbb{R}^2 \to \mathbb{R}$, the corresponding floating-point operation $\circledcirc_p \colon \mathbb{F}_p^2 \to \mathbb{F}_p$ is defined by

$$x \circledcirc_p y := rd_p(x \circ y)\,.$$

The IEEE standard 754 extends this definition to operations with special operands, e.g. $+\infty \oplus_p -\infty :=$ NaN. Note that, due to the rounding, associativity does not hold for floating-point operations, i.e., $(a \circledcirc_p b) \circledcirc_p c$ may differ from $a \circledcirc_p (b \circledcirc_p c)$.

Floating-point arithmetic (FPA) (with precision p) is the logic defined by the structure $\langle \mathbb{F}_p, =, \le, \circledcirc \rangle$. A *term* is an expression over arithmetic operations involving variables or constants over $\mathbb{F}_p$. We also allow conversions between terms with different precision. Given terms t_1, t_2, atoms in FPA are of the form $t_1 \bowtie t_2$ with $\bowtie \in \{=, \le\}$. The Boolean connectives $\wedge$, $\vee$, $\neg$ are used to construct formulas. We consider the combination of FPA with integer bit-vector arithmetic (BVA) and allow both semantic and bit-wise conversions between integer bit-vectors and floating-point bit-vectors. The goal of this work is a decision procedure that determines the satisfiability of FPA+BVA formulas.

III. Propositional Encodings of FPA Formulas

Given a circuit implementation of an IEEE-754 compliant floating-point unit (FPU), each floating-point operation can be modeled as a formula in propositional logic, as we illustrate below. This way, a formula in FPA can — in principle — be translated to an equisatisfiable formula in propositional logic and passed to a SAT-solver to check for satisfiability. This suggests a sound and complete decision procedure for FPA.

[1] For instance, the addition of the binary numbers $1.1 \cdot 2^0 \in \mathbb{F}_1$ and $1.0 \cdot 2^0 \in \mathbb{F}_1$ (1 bit fractional precision) results in $10.1 \cdot 2^0$, which is not representable with 1 bit fractional precision and a mantissa $m < 2$.

The bottleneck is of course the complexity of the resulting propositional formulas, as we demonstrate in the following. Our analysis also hints at sources for *approximating* these formulas in meaningful ways.

A. Addition and Subtraction

Figure 2 shows a high-level description of a floating-point adder/subtracter as implemented in most FPUs. An adder/subtracter is composed of three modules.

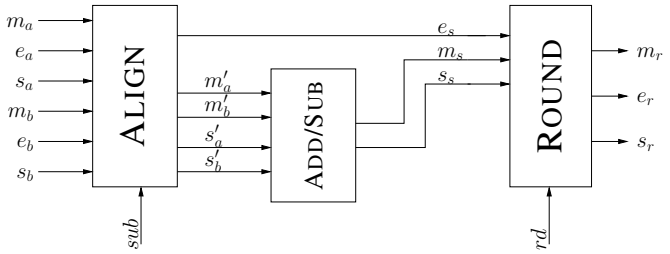

Fig. 2. High-level overview of a floating-point adder/subtracter.

- ALIGN. The mantissa of the smaller operand is shifted by $|e_a - e_b|$ bits to the right, rendering the two exponents equal.
- ADD/SUB. The two resulting mantissas are then added, resp. subtracted, with a standard integer adder.
- ROUND. If the new mantissa has more than p bits, the result is rounded to obtain a number in $\mathbb{F}_p$. The rounding is implemented as a function on the least significant bits of the mantissa.

If $e_a = e_b = e_s = e_r$, the ALIGN module is not needed, since the shift distance is 0. The ROUND module can also be simplified, since the mantissa is not shifted. In this case, the circuit implements *fixed-point* arithmetic: the operation is reduced to the ADD/SUB module. The existence of efficient SAT-encodings for fixed-point arithmetic formulas therefore suggests that reducing the cost of the ALIGN and ROUND modules may improve the performance of a floating-point decision procedure via a SAT-encoding.

Table I shows the number of propositional variables needed for a floating-point adder/subtracter (optimized for propositional SAT, not area or depth), depending on the width p of the mantissa. These numbers confirm that alignment and rounding

Precision	ALIGN	ADD/SUB	ROUND	Total
$p = 5$	295	168	572	1035
$p = 11$	418	252	853	1523
$p = 17$	561	336	1153	2050
$p = 23$	687	420	1447	2554
$p = 29$	813	504	1744	3061
$p = 35$	996	588	2050	3634
$p = 41$	1140	672	2362	4174
$p = 47$	1284	756	2665	4705
$p = 52$	1404	826	2923	5153

TABLE I
NUMBER OF VARIABLES FOR AN FP-ADDER DEPENDING ON p

cause the propositional formula to blow up in size. One way to curb this blow-up is to approximate floating-point operations by reducing the precision p, as we shall do in Section IV.

B. Multiplication and division

A high-level description of a floating-point multiplier/divider is given in Figure 3. Besides the rounder as described above, an FPU implements the following modules for a multiplier/divider:

- ADD/SUB. The exponents of the two operands are first added, for multiplication, resp. subtracted, for division.
- MUL/DIV. The two mantissa are then multiplied, resp. divided.

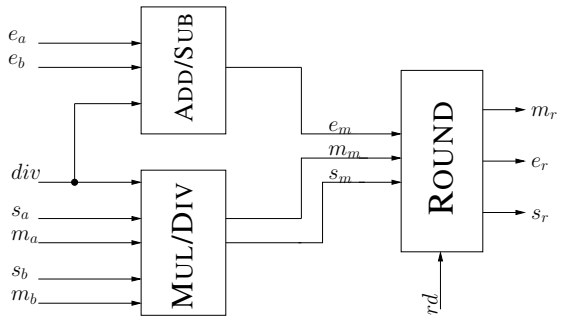

Fig. 3. High-level overview of a floating-point multiplier/divider

Table II shows the number of propositional variables needed for a floating-point multiplier/divider, depending on the width p of the mantissa. As one would expect, the multiplier/divider

Precision	MUL/DIV	ADD/SUB	ROUND	Total
$p = 5$	280	94	674	1048
$p = 11$	982	94	1287	2363
$p = 17$	2188	94	1910	4192
$p = 23$	3898	94	2258	6550
$p = 29$	6112	94	3200	9406
$p = 35$	8830	94	3855	12779
$p = 41$	12052	94	4521	16667
$p = 47$	15778	94	5193	21065
$p = 52$	19268	94	5742	25104

TABLE II
NUMBER OF VARIABLES FOR AN FP-MULTIPLIER DEPENDING ON p

yields a propositional formula that is expensive to decide. One possibility to curb these costs is to reduce the precision p, as this reduces the width of the multiplier. We now discuss how formulas are approximated when p is reduced.

IV. APPROXIMATING FLOATING-POINT ARITHMETIC

Reducing the precision of floating-point operations *approximates* the input formula: a satisfiable formula may become unsatisfiable, or vice versa. In order for the results returned by a SAT solver to still be useful, we need to be aware of the "direction" of the approximation. In this section, we discuss methods for over- and underapproximating floating-point formulas by reducing their precision.

A. Overapproximation

Reducing the precision p of a floating-point operation to p' causes the bits needed for the correct rounding decision to be lost, and the rounding to be based on higher-order bits. It turns out that by making the reduced-precision rounding decision nondeterministic, the reduced-precision formula overapproximates the original.

Definition 3: The *open rounding operation* $\overline{rd}_{p,p'} : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\mathbb{F}_p)$ is defined as

$$\overline{rd}_{p,p'}(X) \;:=\; [\,\lfloor X \rfloor_{p'}, \lceil X \rceil_{p'}\,] \cap \mathbb{F}_p\,,$$

where $\lfloor X \rfloor_{p'} := \min_{x \in X} \lfloor x \rfloor_{p'}$ and $\lceil X \rceil_{p'} := \max_{x \in X} \lceil x \rceil_{p'}$. The set $\overline{rd}_{p,p'}(X)$ can be seen as the smallest precision-p floating-point "interval" Y such that for all $x \in X$, the reduced-precision values $\lfloor x \rfloor_{p'}$, $\lceil x \rceil_{p'}$ are in Y. We use the operator $\overline{rd}_{p,p'}$ to define corresponding open floating-point operations.

Definition 4: For an arithmetic operation $\circ : \mathbb{R}^2 \to \mathbb{R}$, the corresponding *open floating-point operation* $\overline{\circledcirc}_{p,p'} : (\mathcal{P}(\mathbb{F}_p))^2 \to \mathcal{P}(\mathbb{F}_p)$ is defined as:

$$X \,\overline{\circledcirc}_{p,p'}\, Y := \overline{rd}_{p,p'}(\{x \circ y \,|\, x \in X, y \in Y\})\,.$$

The new floating-point operation $\overline{\circledcirc}_{p,p'}$ overapproximates the original operation $\circledcirc_p$ in the sense that $\overline{\circledcirc}_{p,p'}$ yields *more* results than $\circledcirc_p$, i.e., $x \circledcirc_p y \in \{x\}\,\overline{\circledcirc}_{p,p'}\,\{y\}$, for any reduced precision $p' \le p$, as the following lemma shows.

Lemma 1: For p, p' with $p' \le p$, $\;x \circledcirc_p y \in \{x\}\,\overline{\circledcirc}_{p,p'}\,\{y\}$.

Proof: By the definitions of $\circledcirc_p$ and rd_p, $x \circledcirc_p y = rd_p(x \circ y) \in \{\lfloor x \circ y \rfloor_p, \lceil x \circ y \rceil_p\}$. We estimate this set as follows:

$$\begin{array}{lcll} \ldots & \subseteq & [\,\lfloor x \circ y \rfloor_p, \lceil x \circ y \rceil_p\,] \;\cap\; \mathbb{F}_p & \text{[closed interval]} \\ & \subseteq & [\,\lfloor x \circ y \rfloor_{p'}, \lceil x \circ y \rceil_{p'}\,] \;\cap\; \mathbb{F}_p & \text{[nesting prop.]} \\ & = & \overline{rd}_{p,p'}(\{x \circ y\}) & \text{[Def. 3 with } X = \{x \circ y\}] \\ & = & \{x\}\overline{\circledcirc}_{p,p'}\{y\}\,. & \text{[Def. 4]} \end{array}$$

■

One can use the open operations to generate a formula $\overline{\phi}$ that overapproximates the original ϕ. Each floating-point operation $\circledcirc_p$ is replaced by an open version $\overline{\circledcirc}_{p,p'}$, for some reduced precision p'. The reduced precision can be chosen separately for each occurrence of a floating-point operation.

B. Underapproximation

To generate an underapproximation, we devise floating-point operations with *fewer* results than the original operations. Observe that if a floating-point operation with reduced precision p' yields an exact result, then the same result is obtained with the original precision p. Our new floating-point operations are restricted to exact precision p' results. To formalize this idea, we define a modified rounding operator $\underline{rd}_{p,p'}$:

Definition 5: The *no-rounding operator* $\underline{rd}_{p,p'} : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\mathbb{F}_p)$ is defined as

$$\underline{rd}_{p,p'}(X) \;:=\; X \cap \mathbb{F}_{p'}\,.$$

The quantity $\underline{rd}_{p,p'}(\{x\})$ equals $\{x\}$ if no rounding is required to represent x with precision p', i.e., $x \in \mathbb{F}_{p'}$, and the empty set otherwise, independently of p. Floating-point operations $\underline{\circledcirc}_{p,p'}$ that yield exact results only are defined in analogy to Def. 4, with $\overline{rd}$ replaced by $\underline{rd}$. These operations yield fewer results than their original counterparts $\circledcirc_p$. That is, if $\{z\}$ is the result of the new exact operation $\{x\}\underline{\circledcirc}_{p,p'}\{y\}$, then z is also the result of original operation $x \circledcirc_p y$:

Lemma 2: For p, p' with $p' \le p$, $\;\{x\}\underline{\circledcirc}_{p,p'}\{y\} = \{z\}$ implies $x \circledcirc_p y = z$.

Proof: We have $\{x\}\underline{\circledcirc}_{p,p'}\{y\} = \{x \circ y\} \cap \mathbb{F}_{p'} = \{z\}$, thus $z = x \circ y \in \mathbb{F}_{p'} \subseteq \mathbb{F}_p$. From $x \circ y \in \mathbb{F}_p$, we can conclude $\lfloor x \circ y \rfloor_p = \lceil x \circ y \rceil_p = x \circ y = rd_p(x \circ y) = x \circledcirc_p y = z$. ■

One can use $\underline{\circledcirc}$ to generate an *underapproximation* of a formula ϕ, by replacing each floating-point operation with a version that is exact for some reduced precision p'.

In case the constructed underapproximation is shown to be unsatisfiable, nothing can be concluded on the satisfiability of ϕ. One way to resolve the dichotomy between over- and underapproximations is to integrate *both* into an abstraction-refinement framework.

V. The Mixed Abstraction Framework

A. Overview

In Section IV, we have presented over- and underapproximation techniques to simplify a given floating-point formula ϕ. Many existing procedures build *either* over- or underapproximations, depending on whether the goal is to show satisfiability or unsatisfiability. The two types of approximation guarantee a definite decision on the satisfiability of ϕ only in cases that are *orthogonal* for the two types. We therefore propose to combine them in a concerted effort towards analyzing ϕ.

To this end, we propose the abstraction framework shown in Figure 4, which checks the satisfiability of the input formula ϕ. We first identify a set of eligible *transformations*. A transformation is a mapping that turns a FPA formula β into a new one that over- or underapproximates β, for example by replacing some floating-point operation by its open version, as suggested in Section IV. The set of transformations is accordingly partitioned into subsets $Over$ and $Under$. At the beginning of the loop indicated in the figure, an implementation selects some of the eligible transformations and applies them to ϕ in a particular order. Note that the resulting formula ψ in general neither over- nor underapproximates ϕ (hence called "mixed").

Exit points of the loop. Formula ψ is then subject to a satisfiability check. Depending on the outcome of this check, the loop can be exited — with a definite answer — if:

(i) ψ is satisfiable, and the assignment α returned by the solver (suitably extended) satisfies ϕ as well. In this case, the overall answer is "SAT". Or:

(ii) ψ is unsatisfiable, and the resolution proof P returned by the solver is valid for ϕ, too. In this case, the overall answer is "UNSAT".

Refinement. If neither case (i) or (ii) applies, the approximation needs to be refined. This is done by removing some transformations from $Over$ if ψ was found to be spuriously satisfiable, otherwise from $Under$. Which transformations to

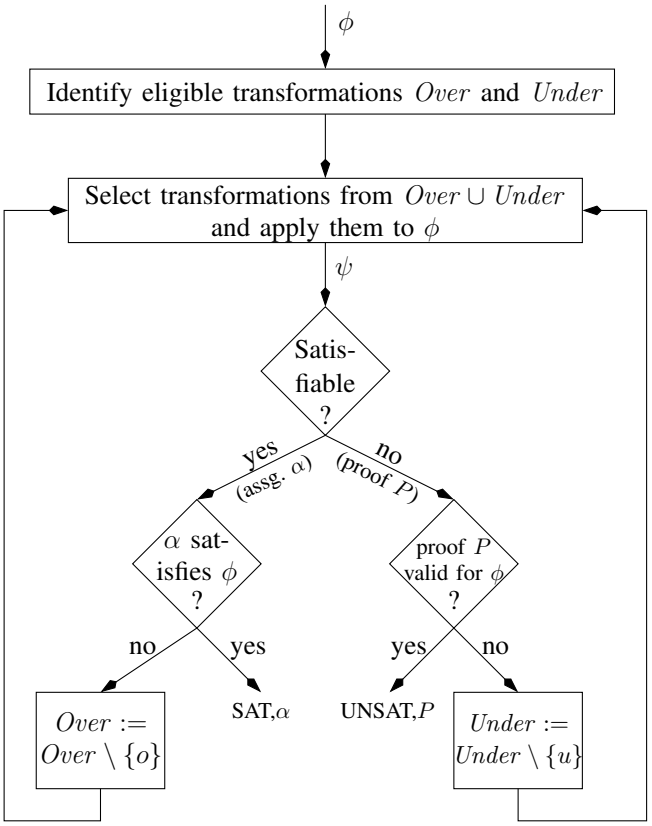

Fig. 4. The Framework of Mixed Abstraction

select for removal is an important implementation decision, which we discuss below in Section VI-B. The loop in Figure 4 is then reentered, and some transformations from the new sets $Over$ and $Under$ are applied to ϕ.

Notice that the procedure can be implemented in a both incremental and backtrackable fashion, provided the underlying SAT solver is incremental and backtrackable.

Property 1: Given a formula ϕ, any algorithm that implements the framework in Figure 4, starting with finite sets $Over$ and $Under$ of transformations, terminates and returns "SAT" if ϕ is satisfiable, "UNSAT" otherwise.

Proof:

(a) Partial correctness: The algorithm outputs "SAT" only in the case that the assignment α was validated successfully against the original formula ϕ. It outputs "UNSAT" only in the case that an unsatisfiability proof for ψ was found to be a valid proof of unsatisfiability for ϕ.

(b) Termination: In each round in which the algorithm does not exit, at least one element is removed from $Over$ or from $Under$. When both sets are exhausted, ψ and ϕ are equivalent, and one of the two exit conditions is trivially satisfied. ■

Note that the correctness property is independent of the distribution and scheduling of over- and underapproximations, and of the strategy for selecting elements o or u to remove in each iteration. Furthermore, it can be shown that finiteness of the sets $Over$ and $Under$ is not even required if one defines a suitable well-quasi-order on the set of all approximation transformations.

The Mixed Abstraction framework generalizes several other abstraction-refinement approaches to satisfiability-checking a formula. The classical CEGAR paradigm *conservatively abstract–check–refine* can be seen as an instance of Figure 4 where the set $Under$ is empty. Therefore, the test whether an unsatisfiability proof for ψ is valid for ϕ can be skipped in favor of an immediate answer "UNSAT". The method presented in [4] has the property that the sequence of approximations obtained from ϕ strictly alternates between strict over- and strict underapproximations. As we experimentally compare this alternating scheme to our own implementation of mixed abstraction, we sketch first how the method of [4] can be applied in a floating-point environment.

VI. Implementing Mixed Abstractions

A. Alternating Abstractions for Floating-Point Arithmetic

The alternating approach by Bryant et al. [4] to integrating over- and underapproximations was implemented for integer bit-vector arithmetic formulas. If the SAT-check on an approximated formula is inconclusive, information obtained from the SAT checker is used to generate a refined approximation of the opposite type, and the procedure repeats.

We can use the approximation methods presented in Section IV to apply the alternating approach to floating-point arithmetic, as follows. We begin with an overapproximation $\overline{\phi}$ of ϕ. To obtain $\overline{\phi}$, the transformations in $Over$ replace all floating-point operations $\circledcirc$ by $\overline{\circledcirc}_{p,p'}$, for some initial reduced precision p'. Since $Under$ is not applied, $\overline{\phi}$ is an overapproximation.

If $\overline{\phi}$ is unsatisfiable the procedure terminates. Otherwise, the decision procedure yields an assignment α. If α also satisfies ϕ, the procedure halts and returns α as a witness. If α is spurious, we extract from it the operands a, b, and the result r of each occurrence of $\overline{\circledcirc}_{p,p'}$. In case $r \neq \circledcirc_p$ we conclude that r is a spurious result and we refine (increase) the precision of $\overline{\circledcirc}_{p'}$; otherwise p' is left unchanged.

Next, the decision procedure builds a refined underapproximation $\underline{\phi}$ as explained in Section IV-B. In this iteration, the transformations in $Under$ replace all occurrences of $\circledcirc$ by $\underline{\circledcirc}_{p,p'}$ for the refined precisions p'; no transformation from $Over$ is applied. In case $\underline{\phi}$ is satisfiable, the procedure terminates and returns α as an assignment for ϕ. Otherwise, the decision procedure yields an UNSAT proof P for $\underline{\phi}$. If P is also a proof for ϕ, the procedure halts and returns P. Otherwise, it checks whether the constraint $X \cap \mathbb{F}_p$ of an exact operation $\underline{\circledcirc}_{p'}$ (Definition 5) is contained in P. If it is, the precision is increased; otherwise it is left unchanged. The next iteration constructs a refined overapproximation. Altogether, this yields a sound and complete decision procedure alternating between over- and underapproximations.

The problem with this approach is that the alternating schedule of over- and underapproximations often leads to ineffective approximations, as some formulas are not amenable to effective overapproximations, while others do not permit

effective underapproximations. As an example, consider the non-associativity formula $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$. This formula is satisfiable, as floating-point addition $\oplus$ is not associative. Satisfiability cannot be proved using an overapproximation. On the other hand, every strict underapproximation of this formula turns out to be unsatisfiable. Thus, this formula cannot be decided using **either** strict over- **or** strict underapproximations.

Our experimental results (Section VII) confirm these predictions for realistic formulas. The lesson is that an implementation of Mixed Abstraction should not be "forced" to apply either type of abstraction for pure schedule reasons. Instead, the structure of the formula itself should dictate how to approximate it. This leads to our approach of "genuinely mixed abstractions", which we present in the following.

B. Genuinely Mixed Abstractions

We now detail our implementation of the mixed abstraction framework. The selection of abstractions is determined by the structure of the formula, and which approximations are most effective on it. We begin with both a very coarse overapproximation and a strong underapproximation: the result of the operation is completely nondeterministic and forced to zero at the same time. Depending on the outcome of the satisfiability check of ψ, either one of these approximations is refined, gradually lifting constraints or gradually increasing the precision of the operator.

The simulation of α on ϕ in the left branch in Figure 4 can be preceded by a check whether any transformation in $Over$ was applied to ϕ. If not, ψ is guaranteed to be an underapproximation of ϕ, and "SAT" and α are returned immediately. Otherwise, α is suitably extended to an assignment for ϕ and checked for satisfaction of ϕ, as suggested by the figure.

A simple and efficient pre-check whether an unsatisfiability proof P for ψ is extendable to ϕ can be performed by computing the set $Var(P) \cap Var(Under)$. This set contains the variables occurring in (the clauses of) the proof P that are involved in any of the underapproximation transformations in $Under$. If empty, we conclude that the underapproximation transformations applied to ϕ are not responsible for the unsatisfiability result for ψ, which hence applies to ϕ, too. The emptiness test can obviously be optimized by checking first whether any transformation in $Under$ was applied to ϕ.

If none of the exit tests succeeds, the algorithm selects approximation transformations o or u to remove at the end of the loop body, in order to refine the approximation. Let us look first at the case that ψ was found to be satisfiable, but the assignment α is spurious (does not satisfy ϕ). If there exists a transformation $o \in Over$ such that α does not satisfy the formula obtained by simplifying ϕ using all transformations *except* o, we can select such an o: its removal guarantees that the spurious assignment α disappears in the next round. If there is no such transformation, we select some o that affects the variables occurring in the spurious assignment α.

If ψ was found to be unsatisfiable, the algorithm computes the set $Var(P) \cap Var(Under)$, as discussed above. If this set is non-empty, then there exists at least one $u \in Under$ such that $Var(u) \cap Var(Under)$ is non-empty. The algorithm picks such an element u for removal from $Under$.

After refining the sets $Over$ and $Under$ as described, Figure 4 suggests to apply the refined (and smaller) sequence of approximations to ϕ again. In practice, approximations selected for removal in the previous step are revoked, so that only local modifications to ψ are necessary. This allows the subsequent satisfiability check to be done incrementally, without restarts of the SAT solver.

VII. Experimental Results

A. Implementation and Benchmarks

We have implemented the algorithm proposed in this paper in combination with a standard bit-flattening decision procedure for integer bit-vector arithmetic. The procedure supports all operators required to model ANSI-C programs. The procedure is fully incremental: the clauses for those parts of the encoding that are not modified, and any clauses learned from those, are retained between iterations. We use MiniSAT2 [6] as our SAT solver. The performance of the integer bit-vector decision procedure we use is comparable to that of a state-of-the-art SMT-$\mathcal{BV}$ solver.

The benchmarks we use are derived from a variety of publicly-available C programs, selected from the SNU real-time [7] and the Mediabench benchmarks [8]; the programs we have selected make extensive use of single- or double-precision floating-point arithmetic. Our encoding is able to support changes of the rounding mode during the program execution, as the rounding mode may be set separately for each individual operator in the formula. However, none of the benchmark programs makes use of this feature, and we have used "round to nearest even" uniformly. We have not observed a significant impact of the specific rounding mode on the performance of either the full encoding or the abstraction-refinement procedure.

In order to obtain verification conditions, we have manually annotated our benchmarks with properties in the form of arithmetic assertions; a few of the programs already contain some (light-weight) assertions. The main property we check is the possibility of arithmetic exceptions, as defined in the IEEE floating-point arithmetic standard. These properties turn out to be difficult; an instance of a counterexample is arithmetic underflowing to zero and a subsequent division of two such numbers, which results in NaN. Such counterexamples can only be obtained if encoding artifacts such as denormal numbers are modelled in a precise manner.

After the annotation, we pass the programs to CBMC [9], which generates a total of 119 FPA decision problems. A satisfying assignment for such a formula corresponds to a counterexample that demonstrates that an assertion can be violated. Our experiments were performed on a machine running Linux on an Intel Xeon CPU with 3 GHz and 4 MB cache.

B. Results

Figure 5 compares the runtime of the mixed abstraction with the full flattening, for a timeout of 1000 s and a memory

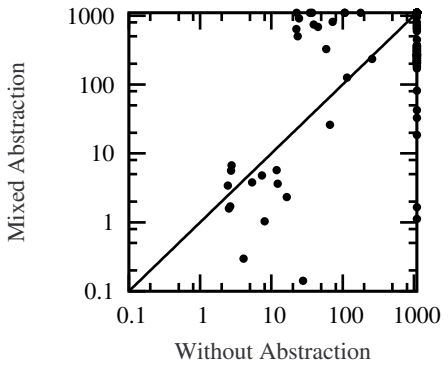

Fig. 5. Runtime comparison of full flattening and the mixed abstraction

limit of 2 GB. The mixed abstraction is not dominating; in some cases (6 out of 119), the final abstraction contains almost all operators at full precision, and as a result, the immediate full flattening is faster. On the other hand, there are 58 large instances for which the abstraction-refinement procedure terminates within the timeout, whereas the full flattening aborts due to excessive memory consumption.

Let us look at some benchmarks in more detail. For a timeout of 7 hours, Table III provides a comparison of the performance of three methods: the full flattening, an alternating abstraction as proposed in [4], and the mixed abstraction. The entries illustrate particular weaknesses and strengths of the three procedures. First of all, even small programs may result in decision problems that are simply too large (or too hard) for a full flattening. Similarly, there are instances in which the alternating abstraction is unable to proceed beyond the first iteration, as either the pure under- or overapproximation is already too hard. The mixed abstraction terminates on a number of instances that are too hard for the other procedures, but typically requires a large number of iterations. This is owed to the extremely coarse initial abstraction, combining both over- and underapproximations. As long as the intermediate abstractions are solvable, the alternating abstraction may converge quicker to a solution (e.g., consider sqrt.c, claim 2).

VIII. Related work

A popular approach to verifying software with floating-point operations is to use *proof assistants*, i.e., programs that prove theorems under the guidance of a human expert. A variety of assistants have been used to prove upper bounds for the deviation from the *real* arithmetic result of a calculation. Examples include proofs using HOL [10], HOL-Light [11]–[13] and ACL2 [14], [15]. However, in case no proof is found, proof assistants typically return little evidence as to whether the proof goal was actually invalid, or whether the proof strategy was too weak to establish its validity.

An alternative approach is to use *abstract interpretation* [2] together with domains that can soundly approximate floating-point computations for classical static analysis. The static analyzer ASTRÉE [16] uses a number of domains which can soundly abstract floating-point computations, including intervals, octagons, polyhedra, and ellipsoid domains.

The adaptation of abstract domains for floating-point numbers is a non-trivial problem due to issues of rounding, the possibility of overflows and underflows, and division by zero errors. Relational abstract domains such as the octagon domain rely on associativity and distributivity of arithmetic operations. These properties do not hold for floating-point numbers. In ASTRÉE, floating-point expressions are therefore approximated by linear expressions over the real field with interval coefficients [17] before they are transformed into their target abstract domains. In [18], a floating-point polyhedra abstract domain based on these ideas is presented. Their linearisation technique is implemented in the APRON library for static analysis [19], which provides sound handling of floating-point expressions for a number of abstract domains.

The propagation of floating-point rounding errors has also been studied extensively in the framework of abstract interpretation [20]–[23]. Such analyses allow to quantify deviations of floating-point computations from their exact result in arithmetic over the reals. Verifying floating-point programs using abstract interpretation shares with our method the advantage of being fully automatic. However, in case the property does not hold, it is usually difficult to obtain a counterexample that can be reproduced on the actual program.

Neither interactive theorem provers nor the use of abstract domains enjoy the characteristics of a *decision procedure*, namely completeness and full automation in deciding floating-point expressions. There has been some work on decision procedures for floating-point arithmetic in the field of *constraint satisfaction programming (CSP)*. Solvers for CSP instances containing floating-point constraints have been applied to automated test-vector generation [24], [25]. This approach combines filtering the possible values of variables using interval techniques with a search procedure for finding actual floating-point values inside these intervals. Their algorithm is mainly geared towards test-vector generation, not verification. The authors state that in some cases where the calculated intervals overapproximate the concrete variable values too coarsely, their approach is unable to terminate with an answer in reasonable time. This includes the case where no such concrete solutions exist.

In this paper, we are interested in verification of software using floating-computations. A different, but related field of research is the verification of floating-point hardware. The work described in [26] is an example of such an approach and provides an overview over the relevant literature.

IX. Conclusion

We have presented an algorithm for iteratively approximating a complex formula by mixing both under- and over-approximations to obtain a formula ψ. In contrast to prior work, ψ need not be an over- nor an underapproximation and can therefore be constructed in a way that yields formulas that are easy to solve. Experimental results indicate improved

Benchmark	Lines of Code	Satisfiable?	No Abstr.	Alternating [4]		Mixed	
			time (s)	time (s)	#iter.	time (s)	#iter.
qurt.c, claim 1	109	no	25	15	1	2	15
qurt.c, claim 2	109	no	25	15	1	0.6	7
qurt.c, claim 3	109	no	25	15	1	1	13
qurt.c, claim 4	109	no	OM	OM	1	478	103
qurt.c, claim 5	109	no	25	15	1	1.2	15
qurt.c, claim 6	109	no	25	15	1	0.6	7
qurt.c, claim 7	109	no	6716	OM	1	84	86
sqrt.c, claim 1	51	no	24	TO	3	13589	44
sqrt.c, claim 2	51	yes	9	608	2	TO	107
minver.c, claim 1	156	no	1	1	1	0.1	1
minver.c, claim 2	156	yes	2	2	2	0.1	1
sin.c, claim 1	46	no	13864	1892	1	281	47
sin.c, claim 2	46	no	13831	1894	1	281	47
sin.c, claim 3	46	no	TO	TO	3	1074	63
gaussian.c, claim 1	108	no	TO	TO	1	14437	137

TABLE III
COMPARISON OF FULL FLATTENING TO ALTERNATING AND MIXED ABSTRACTION

robustness compared to a plain flattening and to an abstraction-refinement scheme based on an alternation of over- and underapproximations.

The algorithm supports incremental solving, is complete, and produces witnesses. In particular, the ability to generate counterexamples for invalid specifications is essential not only for debugging, but also for the high-impact field of *automated test-vector generation*, where counterexamples to carefully crafted specifications translate into test cases meeting certain coverage criteria.

One aspect of future work is how to vary the exponent width r in order to approximate a floating-point operation. While the operations on the exponent contribute only the smaller part of the propositional encoding, many programs exist that only exercise a very small range of exponent values.

REFERENCES

[1] Gander, W.: Heisenberg effects in computer-arithmetic (2005) http://www.inf.ethz.ch/news/focus/res_focus/april_2005.

[2] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL). (1977) 238–252

[3] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification (CAV). (2000) 154–169

[4] Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding bit-vector arithmetic with abstraction. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2007) 358–372

[5] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003) 118–149

[6] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing (SAT). (2003) 502–518

[7] SNU Computer Architecture and Network Laboratory: SNU Real-Time Benchmarks. (1995) http://archi.snu.ac.kr/realtime/benchmark.

[8] Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In: International Symposium on Microarchit (MICRO). (1997) 330–335

[9] Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2004) 168–176

[10] Carreno, V.A.: Interpretation of IEEE-854 floating-point standard and definition in the HOL system. Technical report, NASA Langley Research Center (1995)

[11] Harrison, J.: Formal verification of square root algorithms. Form. Methods Syst. Des. **22** (2003) 143–153

[12] Harrison, J.: Formal verification of floating point trigonometric functions. In: Formal Methods in Computer-Aided Design (FMCAD). (2000) 217–233

[13] Harrison, J.: Floating point verification in HOL light: The exponential function. In: Algebraic Methodology and Software Technology (AMAST). (1997) 246–260

[14] Russinoff, D.M.: A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. Form. Methods Syst. Des. **14** (1999) 75–125

[15] Russinoff, D.M.: A case study in fomal verification of register-transfer logic with ACL2: The floating point adder of the AMD AthlonTM processor. In: Formal Methods in Computer-Aided Design (FMCAD). (2000) 3–36

[16] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation (PLDI). (2003) 196–207

[17] Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: European Symposium on Programming (ESOP). (2004) 3–17

[18] Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Asian Symposium on Programming Languages and Systems (APLAS). (2008) 3–18

[19] Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: Computer Aided Verification (CAV). (2009, to appear)

[20] Goubault, E.: Static analyses of the precision of floating-point operations. In: Static Analysis (SAS). (2001) 234–259

[21] Goubault, E., Martel, M., Putot, S.: Asserting the precision of floating-point computations: A simple abstract interpreter. In: European Symposium on Programming (ESOP). (2002) 209–212

[22] Martel, M.: Static analysis of the numerical stability of loops. In: Static Analysis (SAS). (2002) 133–150

[23] Putot, S., Goubault, E., Martel, M.: Static analysis-based validation of floating-point computations. In: Numerical Software with Result Verification. (2003) 306–313

[24] Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: Principles and Practice of Constraint Programming (CP). (2001) 524–538

[25] Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Softw. Test., Verif. Reliab. **16** (2006) 97–121

[26] Jacobi, C., Berg, C.: Formal verification of the VAMP floating point unit. Formal Methods in System Design **26** (2005) 227–266