Combining Testing and Static Analysis to Overflow and Roundoff Error Detection

Do Thi Bich Ngoc · Mizuhito Ogawa.

Abstract This paper proposes a technique for automatic detection of overflow and roundoff errors, caused by the floating-point number to fixed-point number conversion. First, a new range representation, "extended affine interval", is proposed to overapproximate overflow and roundoff errors. Second, the overflow and roundoff error analysis problem is encoded as a weighted model checking, which is implemented as a static analyzer CANA. Last, we propose a new testing refinement loop, called "counterexample-guided narrowing", by combining the static analysis and testing. They are composed and implemented in a prototype tool, CANAT, in which analysis results are used not only for possible roundoff error detection, but also for finding dominant error factors in input parameters. To avoid widening, currently we focus on programs with bounded loops and arrays with fixed length, which typically appear in encoder/decoder reference algorithms. Experimental results on small programs show that the extended affine interval is much more precise than classical interval, and the counterexample-guided narrowing approach outperforms the random testing technique.

Do Thi Bich Ngoc
School of Information Science - Japan Advanced Institute of Science and Technology
Tel.: +84-0761-51-1247
Fax: +84-0761-51-1149
E-mail: dongoc@jaist.ac.jp

Mizuhito Ogawa
School of Information Science - Japan Advanced Institute of Science and Technology
Tel.: +84-0761-51-1247
Fax: +84-0761-51-1149
E-mail: mizuhito@jaist.ac.jp

Keywords Software verification · Static analysis · Testing · Roundoff error · Overflow error · Affine interval · Classical interval

1 Introduction

In the computers, a real number has a finite representation, i.e., either a floating-point number, or a fixed-point number, which may introduce overflow and roundoff errors (ORE). OREs cause tricky behavior; for instane, consider

$$(333.75-a^2)b^6+a^2(11a^2b^2-121b^4-2)+5.5b^8+(a/(2b))$$

for $a = 77617$, $b = 33096$. This is known as Rump's example [31] and IEEE 754 standard floating operations [22] return the results

Single precision	1.172604
Double precision	1.1726039400531786
Fourfold precision	1.172603940053178631858834904520118 38

which seem that the single precision is enough. However, if we symbolically compute with rational number representations, it will result $-54767/66192$ (approx. -0.8273960599).

Overflow and roundoff errors problem

A famous example that caused a serious disaster by roundoff errors is "The Patriot Missile Failure" [1].

On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.

[1] http://www.ima.umn.edu/ arnold/disasters/patriot.html

The reason is roundoff error cause by representing the real number (1/10) by the fixed-point number.

$$1/10 = 1/24 + 1/25 + 1/28 + 1/29 + 1/212 + 1/213 + ...$$

which corresponds to the binary expansion of 1/10

$$0.0001100110011001100110011001100...$$

The 24 bit register in the Patriot stored instead

$$0.00011001100110011001100$$

introducing an error of

$$0.0000000000000000000000011001100...$$

(in binary), which is about 0.000000095 in decimal. Multiplying by the number of tenths of a second in 100 hours gives $0.000000095 \times 100 \times 60 \times 60 \times 10 = 0.34$. A Scud travels at about 1,676 meters per second, and thus travels more than an half kilometer in this time.

Another famous example of overflow errors is "The Explosion of the Ariane 5" [2].

The ORE problems have been one of the central issues in the numerical analysis [17,13]. There are three kinds of OREs, caused by:

- real numbers to floating-point numbers conversion,
- real numbers to fixed-point numbers conversion, and
- floating-point numbers to fixed-point numbers conversion.

The first category is most extensively investigated. There are lots of works on mathematical reasoning to estimate OREs, and there is a well-known methodology for precise addition and subtraction, which concern the effect of REs [27]. For nonlinear operations, such as multiplication and division, recently verified numerical computation is evolving.

We focus on ORE problems from a different view point, *static detection*. There are several examples of ORE analyses on the real numbers to floating-point numbers conversion [20,21,33,38], including tool implementations [20,38,34].

We focus on ORE anslyses again from a different view point, the floating-point numbers to fixed-point numbers conversion. Our motivation comes from practical demands in industry. They had troubles on converting a DSP reference algorithm into a hard-wired one, along with the floating-point numbers to fixed-point numbers conversion. Apart from difficulties in hardware encoding [3,5,25,26,37,51,52], OREs may cause visible decoding errors.

[2] http://www.ima.umn.edu/ arnold/disasters/ariane.html

For the floating-point numbers to fixed-point numbers conversion, an ORE analysis was proposed in [9–11] with the same motivation. They used a sophisticated representation, Classical interval (CI) [2,40] and Affine interval (AI) [54,55]. CI is simple but imprecise, because it does not handle correlations between variables. AI introduces symbolic manipulations on noise symbols, to handle correlations between variables. AI arithmetic supplies higher precision, especially in linear operations (e.g., addition, subtraction). However, AI introduces a fresh noise symbol at each nonlinear arithmetic operation, which may lead potential inefficiency. The problem raises a natural question: Can we create a new interval that is simpler yet precise as AI?

We also observe the fact that OREs of the floating-point numbers to fixed-point numbers conversion can be tested, since we can compute both floating-point numbers and fixed-point numbers.

Target programs

Motivated by practical demands, our target programs are reference C algorithms for DSP encoders and DSP decoders.

Our observation on DSP encoders/decoders is that they contain unbounded loops, pointers manipulation, dynamic arrays manipulation only in the outermost interface of large input data (e.g., sound, video). The input data are divided into small pieces and processed by the core algorithm (e.g., Invert Direct Cosine Transform algorithm), which (mainly) consists of loops with a bounded number of iterations and arrays with a fixed size [47]. For instance, in the Mpeg decoder, typical arrays have size 8×8, typical loops are 8×8, and the outermost loop iterates depending on the resolution (Fig. 1)

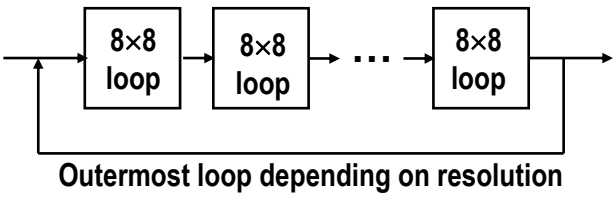


Fig. 1 Typical loops in Mpeg decoder

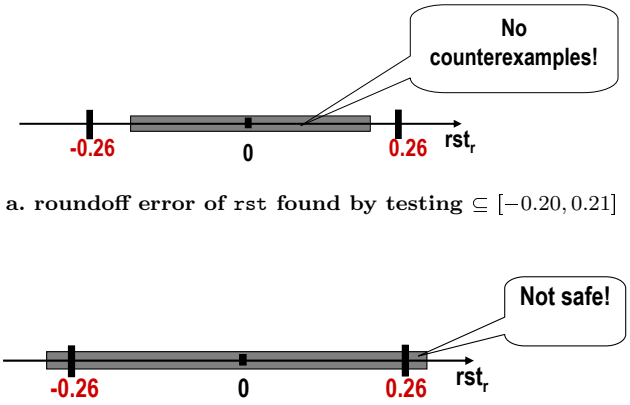
Based on this observation, we restrict targets to a subclass of C programs with bounded loops, fixed size arrays, no pointer manipulations, and no procedure calls.

Then, we set the ORE problems as follows:

Given a program, initial ranges of input parameters, and the fixed-point format,

```
    /* CANAT
    CANAT ALL sign 11 4
    maintest x range -1 3
    maintest y range -10 10
    _test global rst 0.26
    */
    typedef float Real;
    Real rst;
    Real maintest(Real x, Real y){
1.      if (x>0)
2.         {rst=x*x;}
3.      else rst = 3*x;
4.      rst = rst - y;
5.      return rst;
        }
```

Fig. 2 An example of a C program



a. roundoff error of `rst` found by testing $\subseteq [-0.20, 0.21]$

b. roundoff error of `rst` found by analysis $\subseteq [-0.28, 0.28]$

Fig. 3 Results of analyzing and testing C program in Fig. 2

1. *Whether the largest RE of a result lies within given threshold?*
2. *Whether overflow error may occur?*
3. *If they occur, where?*

We say that the program is **"safe"** if for all inputs, REs of the result lie in $[-\theta, \theta]$.

Example 1 Fig. 2 shows a C program with annotations that:

- initial ranges of x, y: $x \in [-1, 3]$, $y \in [-10, 10]$,
- fixed-point format $(11 : 4)$, and
- RE threshold is $\theta = 0.26$

Note that base $b = 2$.

The questions are:

1. Does RE of rst lie within $[-0.26,\ 0.26]$?
2. May overflow error occur? Where?

The OREs will be propagated through computations of the program. Further, the computations themselves cause OREs because the arithmetic needs to round the result to fit the number format. Besides, OREs are also affected by types of statements (e.g., branch, loop, assignment).

Testing vs static analysis

We may face a situation that an ORE analysis reports that the roundoff error of the result exceeds the roundoff error threshold, but a test cannot find any counterexamples.

Example 2 Assume that for the program in Fig. 2, the initial ranges of x, y are $[-1, 3], [-10, 10]$, respectively. The conversion from the floating-point type to fixed-point type such that the width of integer part is 11 and the width of fraction part is 4. It is *safe* if no overflow errors occur and no roundoff errors of *rst* go beyond $[-0.26, 0.26]$.

By random 100 test cases, all roundoff errors lie in the range $[-0.20, 0.21] \subset [-0.26, 0.26]$, which means no counterexamples are found (Fig. 3 a).

The ORE analysis (in Section 4) reports that the roundoff error of `rst` lies in $[-0.28, 0.28]$, which exceeds the roundoff error threshold $[-0.26, 0.26]$ (Fig. 3 b).

Then, both testing and analysis cannot clarify whether the program in Fig. 2 is safe.

The challenge is how to bridge the gap between testing and static analysis?

Main results and paper structure

This paper proposes a technique for automatic detection of overflow and roundoff errors, caused by the floating-point number to fixed-point number conversion. First, inspired by the range representation to under approximate OREs [21], a new range representation, *extended affine interval* (EAI) is proposed to over-approximate OREs. EAI does not increase the number of noise symbols.

Second, the overflow and roundoff error analysis problem is encoded as a weighted model checking, which is implemented as a static analyzer CANA. A weight domain is designed based on the EAI arithmetic.

Last, we propose a new testing refinement loop, called "counterexample-guided narrowing", by combining the static analysis and testing. This refinement of testing is performed by detected OREs by CANA, in which EAI shows dominant error factors among input parameters. They are composed as a prototype tool, CANAT.

To avoid widening, currently we focus on programs with bounded loops and arrays with fixed length, which typically appear in encoder/decoder reference algorithms.

Experimental results on small programs show that the extended affine interval is much more precise than

classical interval, and the counterexample-guided narrowing approach clearly outperforms the random testing technique.

The rest of this paper is organized as follows. Section 2 formally presents ORE problems. In Section 3, we introduce various interval arithemtic. We also propose a new range representation, *EAI*, adding to traditional CI and AI. The ORE analysis based on weighted model checking is introduced in Section 4. We also describe the implementation of the proposed framework in a tool, CANA. Section 5 proposes the counterexample-guided narrowing approach to detect REs and its implementation, CANAT. Section 6 overviews related works. Finally, Section 7 concludes the paper and indicates future works. The content of Section 3 to 4 and that of Section 5 were preliminary reported in [44] and [45], respectively.

2 Representation of Real Numbers and the ORE Problem

We first present overflow and roundoff error (ORE) problem when represent real numbers in computers such as floating-point numbers (Subsection 2.1) and fixed-point numbers (Subsection 2.2). In Subsection 2.3, we introduce ORE arithmetics, which decomposes a number into a pair of floating-point (or fixed-point) and a roundoff error evaluation.

2.1 Floating-point Numbers and ORE problem

2.1.1 Floating-point Numbers

Floating-point numbers are often used to represent real numbers in numerical computation. In a floating-point number, the position of the radix point is dynamic.

Definition 1 A **floating-point number** x has a representation in **base** b, with **sign** s, **significand** m, and **exponent** e, such that

$$x = (-1)^s \times m \times b^e \tag{1}$$

where s is 0 or 1, $m = d_0.d_1...d_{p-1}$ with $0 \leqslant d_i < b$, and e is an integer. The **set of floating-point numbers** is denoted by R_{fl}

Remark 1 In order to optimize the quantity of representable numbers, floating-point numbers are typically in *normalized form*, which puts the radix point after the first non-zero digit (e.i., $d_0 \neq 0$).

Example 3 The **decimal number** $x = 8.75$, represented as $(-1)^0 \times 0.875 \times 10^1$, has $s = 0, m = 0.875, e = 1$. Its equivalent **binary** format is $x = (-1)^0 \times (0.100011) \times 2^{100}$ with $s = 1, m = 0.100011, e = 100$. The corresponding **normal floating-point number** is $x = (-1)^0 \times (1.00011) \times 2^{10}$ with $s = 1, m = 1.00011, e = 10$.

The *floating-point format* $(b, p, emax)$ determines a set of representable floating-point numbers, in which:

- b is base (e.g. 2 or 10)
- p is number of digits in the significand
- $emax$ is the maximum value of exponent e (the minimum value of e is $emin = 1 - emax$).

We basically follow the IEEE7542008 standard [22], shown in Table 1. Thus, a normal floating-point number closest to zero is $\pm b^{emin}$ and a number farest from zero is $\pm(b - b^{1-p}) \times b^{emax}$. For instance, in the binary 64 floating-point format,

- The number closest to zero is

$$\pm 2^{-1022} \approx \pm 2.225073858507202010^{-308}$$

- The number farest from zero is

$$\pm((1-(1/2)^{53})2^{1024}) \approx \pm 1.7976931348623157 \times 10^{308}$$

2.1.2 OREs of Floating-point Numbers

Since floating-point numbers have the finite precision, *roundoff error* may occur due to the finite fraction part, and *overflow error* may occur due to the finite integer part.

Roundoff error (RE) If the significand m of x is represented by more than p bits, x will be truncated (or chopped) in some way. The IEEE7542008 standard defines four rounding algorithms [22].

- *Round to Nearest*: This is the default mode. In this mode results are rounded to the nearest representable value. If the result is midway between two representable values, the even representable is chosen. Even here means the lowest-order bit is zero.
- *Round toward 0*: All results are rounded to the largest representable value whose magnitude is less than that of the result. In other words, if the result is negative it is rounded up; if it is positive, it is rounded down.
- *Round toward* $+\infty$: All results are rounded to the smallest representable value, which is greater than the result.
- *Round toward* $-\infty$: All results are rounded to the largest representable value, which is less than the result.

parameter	binary32	binary64	binary128	decimal32	decimal64	decimal128
b	2	2	2	10	10	10
p	24	53	113	7	16	34
emax	+127	+1023	+16383	+96	+384	+6144

Table 1 The formats of floating-point numbers

Definition 2 Let x be a real number and let x_{fl} be its floating-point number representation. The roundoff error (RE) is $re_{fl}(x) = x - x_{fl}$.

Example 4 In the IEEE 754 decimal32 format ($b = 10, p = 7, emax = 96$),

Floating-point representation For $x = 10/3$, its floating-point number representation is $x_{fl} = (-1)^0 \times 3.3333333 \times 10^0$ and the RE is

$$re_{fl}(x) = 0.000000033333333333333333333333333....$$

Floating-point addition

e = 5	m = 1.234567	(123456.7)
+ e = 2	m = 1.017654	(101.7654)
e = 5	m = 1.234567	
+ e = 5	m = 0.001017654	(after shifting)
e = 5	m = 1.235584654	

This is the exact sum of the operands. It will be rounded to seven digits and then normalized (if necessary). The final result is $e = 5; m = 1.235585$, and the low 3 digits of the second operand (654) are lost. The RE of the addition is $(-1)^0 \times 0.000000654^5 = 0.0654$.

Floating-point multiplication

e = 3	m = 4.734612	
× e = 5	m = 5.417242	
e = 8	m = 25.648538980104	(true product)
e = 8	m = 25.64854	(after rounding)
e = 9	m = 2.564854	(after normalization)

In this case, the lost information of the significand m after normalization are (-0.0000001019896). The RE of the multiplication is

$$(-1)^1 \times 0.0000001019896 \times 10^9 = 101.9896$$

Based on rounding mode, the value of RE may differ.

Lemma 1 *For real number x and its normal floating-point representation x_{fl}, the RE $re_{fl}(x)$ satisfy: $|re_{fl}(x)/x| \leqslant E_{mach}$ where*

- *$E_{mach} = b^{1-p}$ for the rounding toward zero, and*
- *$E_{mach} = b^{1-p}/2$ for the rounding to nearest.*

Overflow error (OE) If the exponents e of x is greater than $emax$, it is an overflow error (OE). More precisely, for a real number x and its floating-point format $(b, p, emax)$, if $x > (b - b^{1-p}) \times b^{emax}$, an OE occurs.

Example 5 In the IEEE 754 decimal32 format ($b = 10, p = 7, emax = 96$),

e = 48	m = 4.734612	
× e = 48	m = 5.417242	
e = 96	m = 25.648538980104	(true product)
e = 96	m = 25.64854	(after rounding)
e = 97	m = 2.564854	(after normalization)

Since $e > emax$, an OE occurs.

2.2 Fixed-point Numbers and ORE Problem

2.2.1 Fixed-point Numbers

Fixed-point numbers are a simple and an easy way to express real numbers, using a fixed number of digits. Due to the hardware simplicity, fixed-point numbers are frequently used when hardware cost, speed, and/or complexity are important issues. Fixed-point places a radix point somewhere in the middle of the digits.

Definition 3 (Fixed-point number) A **fixed-point** number a on base **b** is represented in the form:

$$a = sp a_1 a_2 \dots a_{ip} \cdot a_{ip+1} \dots a_{ip+fp},$$

where

- **sign part** $sp \in \{+, -\}$ determines if a is positive or negative,
- $a_k \in [0, b-1]$ for each $k \in [1, ip + fp]$,
- ip is the width of **integer part**, and
- fp is the width of the **fraction part**.

The set of fixed-point numbers is denoted by R_{fx}. We omit the sign if it is positive.

In the *fixed-point format* (b, ip, fp),

- b is base (e.g. 2 or 10).
- ip is number of digits in the integer part.
- fp is number of digits in the fraction part.

Example 6 The number Π is 3.14159 in the fixed-point format $(b = 10, ip = 2, fp = 5)$.

A fixed-point number has a fixed window of representation. The range value that can be represent is $(-b^{ip} - 1, b^{ip} + 1)$, and the smallest positive is b^{-fp}.

2.2.2 OREs of Fixed-point Numbers

Roundoff error (RE) If the fraction part of number x (real or floating-point) has more than fp digits, it needs to truncates to fit the fixed-point format. This loses information from digits $fp+1$ in fraction part, and an RE occurs.

Definition 4 Let x be a real number and let its fixed-point number representation be x_{fx} under the fixed-point format (b, ip, fp). An RE is $re_{fx}(x) = x - x_{fx}$.

Depending on a rounding mode, the value of RE may differ. For instance, for a real number x and its fixed-point representation x_{fx}, the RE $re_{fx}(x)$ satisfies

- $|re_{fx}(x)| < 2^{-fp}$ for the round toward zero, and
- $|re_{fx}(x)| < b^{-fp}/2$ for the round to nearest.

Example 7 Let the fixed-point format be $(b, ip, fp) = (10, 8, 7)$.

Fixed-point representation $x = 10/3$ is represented as fixed-point number $x_{fx} = +3.3333333$. The RE is

$re_{fx}(x) = x - x_{fx} = 0.000000033333333333333333333....$

Fixed-point multiplication

$$\begin{array}{r} 4.734612 \\ \times\ 5.417242 \\ \hline 25.648538980104 \\ 25.6485389 \qquad \text{(after truncating)} \end{array}$$

It loses information from the 8th digit of the fraction part 0.000000080104, and its RE is 0.000000080104.

Overflow error (OE) If the integer part of number x (real or floating-point) has more than ip digits before the radix, it cannot be represented in the fixed-point numbers, and an OE occurs. More precisely, for a real number x and fixed-point format (b, ip, fp), if $x \geqslant b^{ip}$, an OE occurs.

Example 8 For fixed-point format $(b = 10, ip = 8, fp = 7)$,

$$\begin{array}{r} 4734.612 \\ \times\ 54172.42 \\ \hline 256485389.80104 \end{array}$$

Since the integer part has more than 8 digits, an OE occurs.

2.3 ORE Arithmetic

In a program, the propagated error depends on not only values of variables but also operators of the program. For instance, the result of fixed-point multiplication could potentially have as many bits as the sum of the number of bits in the two operands. An ORE arithmetic decomposes a number into a pair of a finite representation and an RE estimation, and each arithmetic operation is defined on such pairs. There are three kinds of ORE arithmetics corresponding with three kinds of OREs (e.i., real numbers vs floating-point numbers, real numbers vs fixed-point numbers, and floating-point numbers vs fixed-point numbers).

2.3.1 Real-to-Fixed ORE Arithmetic

For a real number x and fixed-point format (b, ip, fp), we denote the **fixed-point part** of x by $rd_{fp}(x)$ and the **RE** by $re_{fp}(x)$ $(= x - rd_{fp}(x))$. If $rd_{fp}(x) > b^{ip}$ we conclude that OE occurs, and if $re_{fp}(x) > \theta$ (where θ is predefined threshold) we conclude that RE occurs. The following definition describes the rules of propagating ORE when converting real numbers to fixed-point numbers.

Definition 5 (Real-to-Fixed ORE arithmetic)
Let $(x_f,\ x_r)$ and $(y_f,\ y_r)$ be pairs of fixed-point parts and REs of real numbers x, y. Real-to-Fixed ORE arithmetic $\circledast = \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ is defined below.

$(x_f, x_r) \boxplus (y_f, y_r) =$
$(rd_{fp}(x_f + y_f),\ x_r + y_r + re_{fp}(x_f + y_f))$
$(x_f, x_r) \boxminus (y_f, y_r) =$
$(rd_{fp}(x_f - y_f),\ x_r - y_r + re_{fp}(x_f - y_f))$
$(x_f, x_r) \boxtimes (y_f, y_r) =$
$(rd_{fp}(x_f \times y_f), x_r \times y_f + x_f \times y_r + x_r \times y_r + re_{fp}(x_f \times y_f))$
$(x_f, x_r) \boxdiv (y_f, y_r) =$
$(rd_{fp}(x_f \div y_f),$
$(x_f + x_r) \div (y_f + y_r) - x_f \div y_f + re_{fp}(x_f \div y_f))$

We define the Real-to-fixed ORE comparison operators by comparing the range values of fixed-point representations. For a given fixed-point representation (x_f, x_r), the corresponding range values are $\tilde{x} = [x_f - |x_r|, x_f + |x_r|]$. The results of Real-to-fixed ORE comparison operators may be **true**, **false**, or **unknown**. **Unknown** means that the ranges of a fixed-point expression traverses both **true** and **false** of the condition, and we cannot decide which will hold in real computation. Formally, Real-to-fixed ORE comparison operators are defined as follows:

Definition 6 (Real-to-fixed ORE comparison operators) Let (x_f, x_r), and (y_f, y_r) be pairs of fixed-point parts and REs of real numbers x, y.
$((x_f, x_r) < (y_f, y_r)) =$

$$\begin{cases} \textbf{true} & \text{if } (x_f + x_r < y_f + y_r) \ \wedge \ (x_f < y_f) \\ \textbf{false} & \text{if } (x_f + x_r \geqslant y_f + y_r) \ \wedge \ (x_f \geqslant y_f) \\ \textbf{unknown} & \text{otherwise} \end{cases}$$

$((x_f, x_r) == (y_f, y_r)) =$

$$\begin{cases} \textbf{true} & \text{if } (x_f = y_f \ \wedge \ x_r = y_r) \\ \textbf{false} & \text{if } (x_f, x_r) < (y_f, y_r) \ \vee \ (y_f, y_r) < (x_f, x_r) \\ \textbf{unknown} & \text{otherwise} \end{cases}$$

Other comparison operators, such as $>$, $!=$, are defined using the operators above.

Example 9 Let $x = 34.5678$, $y = 98.76543$. We assume the fixed-point format $(b = 10, ip = 3, fp = 2)$, "round toward $-\infty$", and the RE threshold $\theta = 0.01$. We have:

- The fixed-point value of x is $x_{fx} = 34.56$ and the corresponding RE is $x_r = 0.0078$
- The fixed-point value of y is $y_{fx} = 98.76$ and the corresponding RE is $y_r = 0.00543$

We next show how to evaluate ORE arithmetic:

- **Addition**:

$$\begin{aligned}(x_f, x_r) \boxplus (y_f, y_r) =& (rd_{fp}(34.56 + 98.76), 0.0078 + \\ & 0.00543 + re_{fp}(34.56 + 98.76)) \\ =& (133.32, \ 0.01323)\end{aligned}$$

That means the result of addition is 133.32 and its RE is $0.01323 > \theta$. Thus, an RE is detected.

- **Multiplication**:
$(x_f, x_r) \boxtimes (y_f, y_r) =$
$(rd_{fp}(34.56 \times 98.76), 0.0078 \times 98.76 + 34.56 \times 0.00543$
$+0.0078 \times 0.00543 + re_{fp}(34.56 \times 98.76))$
$= (3413.14, \ 0.963631154)$
That means the result of multiplication is 3413.14 ($> 10^3$) and its RE is $0.963631154 > \theta$. Thus, both an OE and an RE are detected.

2.3.2 Real-to-Float ORE Arithmetic

For a floating-point format $(b, p, emax)$ and a real number x, we denote the **floating-point part** by $rd_{fl}(x)$ and the **RE** by $re_{fl}(x)$ $(= x - rd_{fl}(x))$. If $rd_{fl}(x) > (b - b^{1-p} \times b^{emax}$, we conclude that OE occurs, and if $re_{fl}(x) > \theta$) where θ is predefined threshold) we conclude that RE occurs. The following definition describes the rules of propagating ORE when converting real numbers to floating-point numbers.

Definition 7 (Real-to-Float ORE arithmetic) Let $(x_f, \ x_r)$ and $(y_f, \ y_r)$ be pairs of floating-point parts and REs of real numbers x, y. Real-to-Float ORE arithmetic $\circledast = \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ is defined below.
$(x_f, x_r) \boxplus (y_f, y_r) =$
$(rd_{fl}(x_f + y_f), \ x_r + y_r + re_{fl}(x_f + y_f))$
$(x_f, x_r) \boxminus (y_f, y_r) =$
$rd_{fl}(x_f - y_f), \ x_r - y_r + re_{fl}(x_f - y_f))$
$(x_f, x_r) \boxtimes (y_f, y_r) =$
$(rd_{fl}(x_f \times y_f), x_r \times y_f + x_f \times y_r + x_r \times y_r + re_{fl}(x_f \times y_f))$
$(x_f, x_r) \boxdiv (y_f, y_r) =$
$(rd_{fl}(x_f \div y_f), (x_f + x_r) \div (y_f + y_r) - x_f \div y_f + re_{fl}(x_f \div y_f))$

Real-to-float ORE comparison operators are defined similar Definition 6.

2.3.3 Float-to-Fixed ORE Arithmetic

For a floating-point number x, the floating-point format $(b, p, emax)$, and the fixed-point format (b, ip, fp), we denote the **fixed-point part** by $rd_{fx}(x)$ and the **RE** by $re_{ff}(x)$ $(= re_{fx}(x) - re_{fl}(x))$. If $rd_{fx}(x) > b^{ip}$ we conclude that an OE occurs, and if $re_{fx}(x) > \theta$ (where θ is predefined threshold) we conclude that an RE occurs. The following definition describes the rules of propagating ORE between floating-point numbers and fixed-point numbers.

Definition 8 (Float-to-Fixed ORE arithmetic) Let $(x_f, \ x_r)$ and $(y_f, \ y_r)$ be pairs of fixed-point parts and REs of floating-point numbers x, y. Float-to-Fixed ORE arithmetic $\circledast = \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ is defined below.
$(x_f, x_r) \boxplus (y_f, y_r) =$
$(rd_{fx}(x_f + y_f), \ x_r + y_r + re_{ff}(x_f + y_f))$
$(x_f, x_r) \boxminus (y_f, y_r) =$
$(rd_{fx}(x_f - y_f), \ x_r - y_r + re_{ff}(x_f - y_f))$
$(x_f, x_r) \boxtimes (y_f, y_r) =$
$(rd_{fx}(x_f \times y_f), x_r \times y_f + x_f \times y_r + x_r \times y_r + re_{ff}(x_f \times y_f))$
$(x_f, x_r) \boxdiv (y_f, y_r) =$
$(rd_{fx}(x_f \div y_f),$
$(x_f + x_r) \div (y_f + y_r) - x_f \div y_f + re_{ff}(x_f \div y_f))$

Float-to-fixed ORE comparison operators are defined similar Definition 6.

More precise ORE estimation

When we fix the conversion, such as from the floating-point IEEE 754 binary64 $(2, 53, 1024)$ to the fixed-point $(2, ip, fp)$ with size 2 bytes (e.i., $ip + fp = 16$) (which frequently appears in practice), we can obtain better

estimation of OREs. Assume "round to nearest" in Definition 8.

Let $\delta_\circ = re_{fx}(x \circ y) - re_{fl}(x_f \circ y_f)$ where $\circ \in \{+, -, \times, \div\}$. We now find the bound of $\delta_\circ$ by considering the bound of $re_{fl}(x_f \circ y_f)$ and $re_{fx}(x \circ y)$:

- *Floating-point roundoff error* $rd_{fl}(x \circ y)$:
  Assume $rd_{fl}(x \circ y) = (-(1)^s \times m \times b^e)$, we have $|re_{fl}(x \circ y)| < 2^{-53+e}/2$. Without loss of generality, we can assume $e \leqslant ip$ (otherwise an OE occurs in the fixed-point operator $(x_f \circ y_f)$). Thus, we have:

$$\begin{aligned} |re_{fl}(x \circ y)| &< 2^{-53+e}/2 \\ &< 2^{-53+ip}/2 \\ &< 2^{-53+16-fp}/2 (\text{because } ip + fp = 16) \\ &< 2^{-38-fp} \end{aligned}$$

- *Fixed-point roundoff error* $re_{fx}(x_f \circ y_f)$:
  Because the fixed-point format is unique, the results of the addition and the substraction have the same format. Thus, $re_{fx}(x_f \circ y_f) = 0$ for $\circ \in \{+, -\}$.
  For the multiplication, the fraction part of the result has $2 \times fp$ digits. The fraction part is round to fp digits, and $|re_{fx}(x_f \times y_f)| < 2^{fp}/2 - 2^{2\times fp}/2$.
  For the division, similarly $|re_{fx}(x_f \div y_f)| < 2^{fp}/2$.

Hence, we obtain the **Float64-to-Fixed16 ORE arithmetic** by replacing $re_{ff}(x \circ y)$ with $\delta_\circ$, where $\circ \in \{+, -, \times, \div\}$ and

$$\begin{cases} |\delta_+| < 2^{-38-fp} \\ |\delta_-| < 2^{-38-fp} \\ |\delta_\times| < 2^{fp-1} - 2^{2\times fp-1} + 2^{-38-fp} \\ |\delta_\div| < 2^{fp-1} + 2^{-38-fp} \end{cases} \quad (2)$$

3 Interval Arithmetics

In order to estimate OREs of arithmetic operations, there are two known range representations: classical interval [40] and affine interval [54,55]. In this section, we firstly describe these two methods in detail in Subsections 3.1 and 3.2. Then, inspired by the idea in [21] for under approximation, we propose an "extended affine interval" for overapproximation in Subsection 3.3. Lastly, we represent how to implement these intervals on computers using floating-point type in Subsection 3.4.

3.1 Classical Interval

Classical interval (CI) was introduced in the 1960s by Moore [40] as an approach to putting bounds on rounding errors in mathematical computations. In CI, the upper and the lower bounds describe possible values.

Definition 9 A **classical interval** of x is an interval $\overline{x} = [x_l, x_h]$ with $x_l \leqslant x \leqslant x_h$. The set of classical intervals is denoted by $\overline{R}$.

Definition 10 CI arithmetic consists of operations $\{\overline{+}, \overline{-}, \overline{\times}, \overline{\div}\}$ on pairs of CIs defined below:

$$\begin{aligned} [x_l,\ x_h] \ \overline{+}\ [y_l,\ y_h] &= [x_l + y_l,\ x_h + y_h] \\ [x_l,\ x_h] \ \overline{-}\ [y_l,\ y_h] &= [x_l - y_h,\ x_h - y_l] \\ [x_l,\ x_h] \ \overline{\times}\ [y_l,\ y_h] &= [min(x_l y_l, x_l y_h, x_h y_l, x_h y_h), \\ &\qquad max(x_l y_l, x_l y_h, x_h y_l, x_h y_h)] \\ [x_l, x_h] \ \overline{\div}\ [y_l, y_h] &= [x_l, x_h] \ \overline{\times}\ [\tfrac{1}{y_h}, \tfrac{1}{u_l}] \text{ if } 0 \notin [y_l, y_h] \end{aligned}$$

The following example demonstrates how to compute CI operations:

Example 10 For $x \in \overline{x} = [-1, 3],\ y \in \overline{y} = [-6, 10]$. Let us compute the bound of $z = x \circ y$ ($\circ \in \{+,\ -,\ \times,\ \div\}$) using CI:

- Addition $z = x + y$:

$$\begin{aligned} \overline{z} &= \overline{x} \ \overline{+}\ \overline{y} \\ &= [-1, 3] \ \overline{+}\ [-6, 10] \\ &= [-1 - 6,\ 3 + 10] \\ &= [-7, 13] \end{aligned}$$

- Substraction $z = x - y$:

$$\begin{aligned} \overline{z} &= \overline{x} \ \overline{-}\ \overline{y} \\ &= [-1, 3] \ \overline{-}\ [-6, 10] \\ &= [-1 - 10,\ 3 - (-6)] \\ &= [-11, 9] \end{aligned}$$

- Multiplication $z = x \times y$:

$$\begin{aligned} \overline{z} &= \overline{x} \ \overline{\times}\ \overline{y} \\ &= [-1, 3] \ \overline{\times}\ [-6, 10] \\ &= [min\{6, -10, -18, 30\},\ max\{6, -10, -18, 30\}] \\ &= [-18, 30] \end{aligned}$$

- Division $z = x \div y$:

$$\begin{aligned} \overline{z} &= \overline{x} \ \overline{\div}\ \overline{y} \\ &= [-1, 3] \ \overline{\div}\ [-6, 10] \end{aligned}$$

  Since $0 \in [-6, 10]$, we cannot compute the bound of z; instead a "devision by zero" warning occurs.

For $\overline{x},\ \overline{x}_1,\ ...,\ \overline{x}_n \in \overline{R}$, $\circ \in \{\overline{+}, \overline{-}, \overline{\times}, \overline{\div}\}$, and a constant c, we denote:

- $\overline{x}_1 \overline{x}_2 = \overline{x}_1 \ \overline{\times}\ \overline{x}_2$, $c\overline{x} = \overline{x}c = \overline{x} \ \overline{\times}\ [c,\ c]$,
- $c \circ \overline{x} = [c, c] \circ [x]$, $\overline{x} \circ c = \overline{x} \circ [c, c]$, and
- $\sum_{i=1}^{n} \overline{x}_i = \overline{x}_1 \ \overline{+}\ \overline{x}_2 \ \overline{+}\ \cdots \ \overline{+}\ \overline{x}_n$.

CI arithmetic assumes that all intervals are independent, even if exact values are dependent. The next example illustrates such a problem.

Example 11 Let $x \in \overline{x} = [-1, 3]$. It is easy to see that:

$$\begin{aligned}\overline{x} \mathbin{\overline{-}} \overline{x} &= [-1, 3] \mathbin{\overline{-}} [-1, 3] \\ &= [-4, 4]\end{aligned}$$

CI arithmetic assumes the first operand and the second operand to be independent, while in fact, they represent the same value x and the exact result is $[0, 0]$.

This leads to a great loss of precision in a long computation chain, which is called "error explosion".

3.2 Affine Interval

Affine interval (AI) was introduced by Stolfi [54,55] as a model for self-validated numerical analysis. It was proposed to address the "error explosion" problem in conventional CI. In AI, the value are represented as affine combinations (affine forms) of certain primitive noise symbols, which stand for sources of uncertainty in the data or approximations made during the computation.

Definition 11 An **Affine interval** of x is a formula

$$\ddot{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n$$

with $x \in [x_0 - \sum_i^n |x_i|, x_0 + \sum_i^n |x_i|]$. x_0 is called the **central value**. For each $i \in [1, n]$, $\varepsilon_i \in [-1, 1]$ is a **noise symbol**, which stands for an independent component of the total uncertainty. The set of affine interval forms is denoted by $\ddot{R}$.

In AI arithmetic, the results of linear operations (e.i., addition, subtraction) are straightforward operations on AIs. However, the results of nonlinear operations (e.i., multiplication, division) are not AI forms. Hence, we need to approximate the nonlinear parts of the results by introducing new noise symbols.

Definition 12 AI arithmetic consists of operations $\{\ddot{+}, \ddot{-}, \ddot{\times}, \ddot{\div}\}$ on pairs of AIs as defined below. Let $\ddot{x} = x_0 + \sum_{i=1}^n x_i\varepsilon_i$ and $\ddot{y} = y_0 + \sum_{i=1}^n y_i\varepsilon_i$. AI operations are as defined below:

$$\begin{aligned}
\ddot{x} \mathbin{\ddot{+}} \ddot{y} &= (x_0 + y_0) + \textstyle\sum_{i=1}^n (x_i + y_i)\varepsilon_i \\
\ddot{x} \mathbin{\ddot{-}} \ddot{y} &= (x_0 - y_0) + \textstyle\sum_{i=1}^n (x_i - y_i)\varepsilon_i \\
\ddot{x} \mathbin{\ddot{\times}} \ddot{y} &= x_0 y_0 + \textstyle\sum_{i=1}^n (x_0 y_i + x_i y_0)\varepsilon_i + B\varepsilon_{n+1} \\
\ddot{x} \mathbin{\ddot{\div}} \ddot{y} &= \ddot{x} \mathbin{\ddot{\times}} (\tfrac{1}{\ddot{y}}), \text{if } 0 \notin [x_0 - \textstyle\sum_i^n |x_i|, x_0 + \sum_i^n |x_i|]
\end{aligned}$$

where $\varepsilon_{n+1} \in [-1, 1]$ is a new noise symbol, B is the maximum value of $(\sum_{i=1}^n x_i\varepsilon_i)(\sum_{i=1}^n y_i\varepsilon_i)$, and $\frac{1}{\ddot{y}}$ is computed by Chebyshev approximation [54].

The range of values described an AI is evaluated by replacing each noise symbol ϵ_i with $[-1, 1]$.

The advantage of AI is precision on linear operations, compared to CI. For instance, in Example 11, values in a CI $\overline{x} = [-1, 3]$ are equivalently described by an AI $\ddot{x} = 1 + 2\varepsilon_x$. Then,

$$\ddot{z} = \ddot{x} \mathbin{\ddot{-}} \ddot{x} = 0$$

which shows the exact result of the subtraction $(x - x)$.

For multiplication, there are choices to approximate B, and a direct approximation of B is $(\sum_{i=1}^n |x_i|)(\sum_{i=1}^n |y_i|)$. For division, we apply Chebyshev approximation below.

Chebyshev approximation in division

Chebyshev approximation aims to minimize the maximum absolute error. Let $\mathcal{F}$ be some space of functions, e.g., polynomials, affine forms. An element of $\mathcal{F}$ that minimizes the maximum absolute difference from a given function f over a specified domain Ω is known as a Chebyshev (or minimax) $\mathcal{F}$-approximation to f over Ω. We briefly overview the results in [54].

For univariate functions, the minimax affine approximation is characterized by the following property.

Theorem 1 *[54] Let f be a bounded and continuous function from some closed and bounded interval $I = [a, b]$ to R. Let h be the affine function that best approximates f in I under the minimax error criterion. Then, there exist three distinct points u, v, $w \in I$ where the error $f(x) - h(x)$ has maximum magnitude; and the sign of the error alternates when the three points are considered in ascending order.*

This theorem provides an algorithm for finding the optimum approximation in many cases, via the following corollary:

Corollary 1 *[54] Let f be a bounded and twice differentiable function defined on some interval $I = [a, b]$, whose seconde derivative f'' does not change sign inside I. Let $f^a(x) = \alpha x + \zeta$ be its minimax affine approximation in I. Then:*

- *The coefficient α is simply $(f(b) - f(a))/(b - a)$, the slope of the line $r(x)$ that interpolates the points $(a, f(a))$ and $(b, f(b))$.*
- *The maximum absolute error will occur twice (with the same sign) at the endpoints a and b of the range, and once (with the opposite sign) at every interior point u of I where $f'(u) = \alpha$.*
- *The independent term ζ is such that $\alpha u + \zeta = (f(u) + r(u))/2$, and the maximum absolute error is $\delta = |f(u) = r(u)|/2$.*

This result gives us a method for finding the optimum coefficients α and ζ, as long as we can solve the equation $f'(u) = \alpha$.

If an AI $\ddot{y}$ includes zero, a "division by zero" warning occurs. Thus, we only consider the cases $\overline{y} = [l, h]$ are entirely either positive or negative (i.e., $l > 0$ or $h < 0$). The Chebyshev approximation of $\frac{1}{\ddot{y}}$ (Fig. 4) is computed as follows:

- $a = min\{|l|, |h|\}$, $b = max\{|l|, |h|\}$.
- $\alpha = -1/b^2$.
- $d_{max} = 1/a - \alpha a$, $d_{min} = 1/b - \alpha b$.
- $\zeta = (d_{min} + d_{max})/2$, if $l < 0$ then $\zeta = -\zeta$.
- $\delta = (d_{max} - d_{min})/2$.
- $\frac{1}{\ddot{y}} = \alpha\ddot{y} + \zeta + \delta\varepsilon_k$, where ε_k is a new noise symbol.

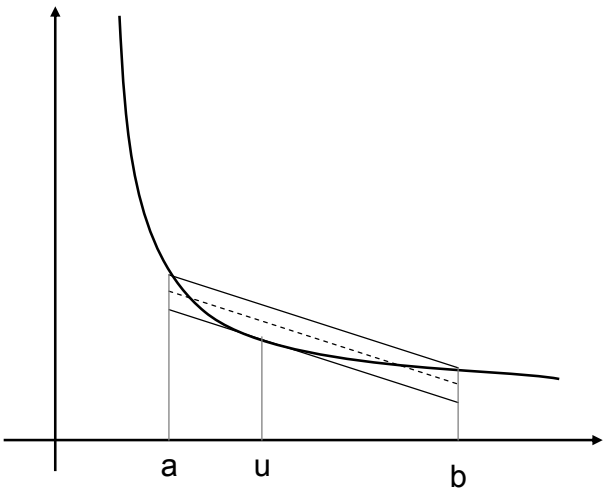


Fig. 4 Chebyshev approximation for $\frac{1}{\ddot{y}}$

Conversion between CI and AI

Standard range representation is CI. To apply AI, conversion between them are needed.

- *CI to AI*: Given a CI $\overline{x} = [l,\ h]$, a corresponding AI is $\ddot{x} = \frac{l+h}{2} + \frac{h-l}{2}\varepsilon_k$. Under valuations of noise symbol ε_k to $[-1, 1]$, they represent the same range. This is called *AI coercion*.
- *AI to CI*: An AI $\ddot{x} = x_0 + \Sigma_{i=1}^{n} x_i\varepsilon_i$ is projected to a CI $\overline{x} = [x_0 - \Sigma_{i=1}^{n}|x_i|,\ x_0 + \Sigma_{i=1}^{n}|x_i|]$. This projection loses information on dependency of uncertainty. This is called *AI projection*.

The following example demonstrates how to propagate the ranges by using AI:

Example 12 For $x \in \overline{x} = [-1, 3],\ y \in \overline{y} = [-6, 10]$. The corresponding AI coercions of $\overline{x},\ \overline{y}$ are:

- $\ddot{x} = 1 + 2\varepsilon_x$
- $\ddot{y} = 2 + 8\varepsilon_y$

Let us compute the bound of $z = x \circ y$ ($\circ \in \{+,\ \times\}$) using AI:

- Addition $z = x + y$:

  $$\begin{aligned}\ddot{z} &= \ddot{x} \mathbin{\ddot{+}} \ddot{y} \\ &= (1 + 2\varepsilon_x) \mathbin{\ddot{+}} (2 + 8\varepsilon_y) \\ &= 3 + 2\varepsilon_x + 8\varepsilon_y\end{aligned}$$

  The AI projection of $\ddot{z}$ is $[3 - 2 - 8, 3 + 2 + 8] = [-7, 13]$.
- Multiplication $z = x \times y$:

  $$\begin{aligned}\ddot{z} &= \ddot{x} \mathbin{\ddot{\times}} \ddot{y} \\ &= (1 + 2\varepsilon_x) \mathbin{\ddot{\times}} (2 + 8\varepsilon_y) \\ &= 2 + 4\varepsilon_x + 8\varepsilon_y + 16\varepsilon_1\end{aligned}$$

  where ε_1 is new noise symbol standing for $\varepsilon_x\varepsilon_y$. The AI projection of $\ddot{z}$ is $[2 - 4 - 8 - 16, 2 + 4 + 8 + 16] = [-26, 30]$.

3.3 Extended Affine Interval

AI is more precise than CI for linear operations, but each time we perform a nonlinear operation, it introduces a new noise symbol. This would be problematic for a program with a large number of nonlinear operations.

In [21], instead of introducing new noise symbols, coefficients of noise symbols are replaced with CIs. Arithmetic operations are designed for under approximation, and we apply similar ideas for overapproximation. This is called an *extended affine interval* (EAI), which also avoids introduction of new noise symbols for nonlinear operations.

Definition 13 An **extended affine interval** of x is a formula

$$\widehat{x} = \overline{x}_0 \mathbin{\overline{+}} \sum_{k=1}^{n} \overline{x}_k\varepsilon_k$$

with $x \in \overline{x}_0 \mathbin{\overline{+}} \sum_{k=1}^{n} \overline{x}_k[-1, 1]$, where $\varepsilon_i \in [-1, 1]$ is a noise symbol for each $i \in [1, n]$ and $\overline{x}_j \in \overline{R}$ for each $j \in [0, n]$. The set of extended affine intervals is denoted by $\widehat{R}$.

The linear operations of EAI arithmetic are designed similarly to those of AI arithmetic. For nonlinear operations, unlike AI, EAI arithmetic does not need to introduce new noise symbols. The results of nonlinear operations approximate nonlinear parts, with CI coefficients. For example, let us consider the multiplication of two EAIs. Let $\widehat{x} = \overline{x}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{x}_i\varepsilon_i,\ \widehat{y} =$

$\overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^n \overline{y}_i\varepsilon_i$. Without loss of generality, assume that $\sum_{k=1}^n \overline{y}_k[-1,1] \sqsubseteq \sum_{k=1}^n \overline{x}_k[-1,1]$. We have:
$\hat{x} \mathbin{\hat{\times}} \hat{y} = (\overline{x}_0 \mathbin{\overline{+}} \sum_{i=1}^n \overline{x}_i\varepsilon_i) \mathbin{\hat{\times}} (\overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^n \overline{y}_i\varepsilon_i)$
$= \overline{x}_0\overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^n (\overline{x}_0\overline{y}_i \mathbin{\overline{+}} \overline{x}_i\overline{y}_0 \mathbin{\overline{+}} \overline{x}_iB)\varepsilon_i$, where $B = \sum_{i=1}^n \overline{y}_i\varepsilon_i$. A direct approximation of B is $\sum_{k=1}^n \overline{y}_k[-1,1]$. Formally, EAI arithmetic is defined as follows:

Definition 14 EAI arithmetic consists of operations $\{\hat{+}, \hat{-}, \hat{\times}, \hat{\div}\}$ on pairs of EAIs.

Let $\hat{x} = \overline{x}_0 \mathbin{\overline{+}} \sum_{i=1}^n \overline{x}_i\varepsilon_i$, $\hat{y} = \overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^n \overline{y}_i\varepsilon_i$, $\overline{X} = \sum_{k=1}^n (\overline{x}_k\ [-1,1])$, and $\overline{Y} = \sum_{k=1}^n (\overline{y}_k\ [-1,1])$. Then,

$$\begin{array}{l}
\hat{x} \mathbin{\hat{+}} \hat{y} = (\overline{x}_0 \mathbin{\overline{+}} \overline{y}_0) \mathbin{\overline{+}} \sum_{i=1}^n (\overline{x}_i \mathbin{\overline{+}} \overline{y}_i)\varepsilon_i \\
\hat{x} \mathbin{\hat{-}} \hat{y} = (\overline{x}_0 \mathbin{\overline{-}} \overline{y}_0) \mathbin{\overline{+}} \sum_{i=1}^n (\overline{x}_i \mathbin{\overline{-}} \overline{y}_i)\varepsilon_i \\
\hat{x} \mathbin{\hat{\times}} \hat{y} = \overline{x}_0\overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^n (\overline{x}_0\overline{y}_i \mathbin{\overline{+}} \overline{x}_i\overline{y}_0)\varepsilon_i \mathbin{\overline{+}} B \\
\hat{x} \mathbin{\hat{\div}} \hat{y} = \hat{x} \mathbin{\hat{\times}} (\frac{1}{\hat{y}}) \quad \text{if } 0 \notin \overline{x}_0 \mathbin{\overline{+}} \sum_{k=1}^n \overline{x}_k[-1,1]
\end{array}$$

where:

$$B = \begin{cases} (\sum_{i=1}^n \overline{x}_i\varepsilon_i)\overline{Y} & \text{if } \overline{Y} \subseteq \overline{X} \\ \overline{X}(\sum_{i=1}^n \overline{y}_i\varepsilon_i) & \text{otherwise} \end{cases}$$

and $\frac{1}{\hat{y}}$ is computed by Chebyshev approximation [54].

Similar to AI arithmetic, the commutative property holds for both addition and multiplication; the associative property only holds for addition; and the distributive property does not hold.

Remark 2 The overapproximation B may conceal some noise symbols. If we are sensitive to this matter, B can be modified as:

$$B = \alpha(\sum_{i=1}^n \overline{x}_i\varepsilon_i)\overline{Y} \mathbin{\overline{+}} \beta\overline{X}(\sum_{i=1}^n \overline{y}_i\varepsilon_i)$$

with $\alpha = \frac{|\overline{X}|}{(|\overline{X}| \mathbin{\overline{+}} |\overline{Y}|)}$ and $\beta = (1-\alpha)$.

Conversion between CI and EAI

To apply EAI, the conversion between them are needed.

- *CI to EAI*: Given a CI $\overline{x} = [l,\ h]$, a corresponding EAI is $\hat{x} = \frac{l+h}{2} \mathbin{\overline{+}} \frac{h-l}{2}\varepsilon_k$. Under valuations of a noise symbol ε_k to $[-1,1]$, they represent the same range. This is called *EAI coercion*.
- *EAI to CI*: An EAI $\hat{x} = \overline{x}_0 \mathbin{\overline{+}} \Sigma_{i=1}^n \overline{x}_i\varepsilon_i$ is projected to a CI $\overline{x} = \overline{x}_0 \mathbin{\overline{+}} \Sigma_{i=1}^n \overline{x}_i[-1,\ 1]$. Replacement of a noise symbol ε_k to $[-1,1]$ loses information on dependency of uncertainty. This is called *EAI projection*.

Example 13 For $x \in \overline{x} = [-1,3],\ y \in \overline{y} = [-6,10]$. The corresponding EAI coercions of $\overline{x},\ \overline{y}$ are:

- $\hat{x} = 1 \mathbin{\overline{+}} 2\varepsilon_x$
- $\hat{y} = 2 \mathbin{\overline{+}} 8\varepsilon_y$
- Addition $z = x + y$:

  $$\begin{array}{l} \hat{z} = \hat{x} \mathbin{\hat{+}} \hat{y} \\ \quad = (1 \mathbin{\overline{+}} 2\varepsilon_x) \mathbin{\hat{+}} (2 \mathbin{\overline{+}} 8\varepsilon_y) \\ \quad = 3 \mathbin{\overline{+}} 2\varepsilon_x \mathbin{\overline{+}} 8\varepsilon_y \end{array}$$

  The EAI projection of $\hat{z}$ is

  $$3 \mathbin{\overline{+}} 2[-1,1] \mathbin{\overline{+}} 8[-1,1] \ = \ [-7,13]$$

- Multiplication $z = x \times y$:

  $$\begin{array}{l} \hat{z} = \hat{x} \mathbin{\hat{\times}} \hat{y} \\ \quad = (1 \mathbin{\overline{+}} 2\varepsilon_x) \mathbin{\hat{\times}} (2 \mathbin{\overline{+}} 8\varepsilon_y) \\ \quad = 2 \mathbin{\overline{+}} 4\varepsilon_x \mathbin{\overline{+}} 8\varepsilon_y \mathbin{\overline{+}} B \end{array}$$

  where $\overline{X} = 2[-1,1] = [-2,2]$ and $\overline{Y} = 8[-1,1] = [-8,8]$.

  Because $\overline{X} \subset \overline{Y}$. Hence, $B = 8\varepsilon_y\overline{X} = [-16,16]\varepsilon_y$. Then, $\hat{z} = 2 \mathbin{\overline{+}} 4\varepsilon_x \mathbin{\overline{+}} [-8,24]\varepsilon_y$

  The EAI projection of $\hat{z}$ is

  $$2 \mathbin{\overline{+}} 4[-1,1] \mathbin{\overline{+}} [-8,24] \overline{\times} [-1,1] \ = \ [-26,30]$$

Although EAI does not introduce new noise symbols, this does not mean EAI arithmetic is always less precise than AI arithmetic. AI arithmetic only advances in cases when we reuse the results of nonlinear parts.

Example 14 Let $z = x \times x; t = z - z$ and $x \in [-1,31]$.

- AI arithmetic: $\ddot{x} = 1 + 2\varepsilon_x,\ \ddot{z} = 1 + 4\varepsilon_x + 4\varepsilon_1$ where ε_1 is introduced for multiplication $\varepsilon_x\varepsilon_x$.
  $\ddot{t} = \ddot{z} \ddot{-} \ddot{z} = 0$
- EAI arithmetic: $\hat{x} = 1\overline{2}\varepsilon_x,\ \hat{z} = 1\overline{+}[0,8]\varepsilon_x$.
  $\hat{t} = \hat{z} \hat{-} \hat{z} = [-8,8]\varepsilon_x$

In this case, AI is more precise than EAI. However, if we compute the bound of $t = x \times x - x \times x$ (without reusing the multiplication $x \times x$), both AI and EAI return the same bound.

3.4 Interval Representations by Floating-point Numbers

As pointed in [54,55], the interval representations themselves are affected by OREs, since boundaries and coeffcients are represented by floating-point numbers. We will briefly overview how to overapproximate CI, AI, and EAI.

For $x \in R$, we define:

- $\downarrow x \in R_{fl}$ is the round toward $+\infty$ of x,
- $\uparrow x \in R_{fl}$ is the round toward $-\infty$ of x, and

Floating-point classical interval

For CI, a safe approximation is to truncate down the lower bound and truncate up the upper bound of an interval.

Definition 15 A *floating-point classical interval* of CI $\overline{x} - [l, h]$ for $l, h \in R$ is

$$\updownarrow \overline{x} = \updownarrow [l, h] = [\downarrow l, \uparrow h].$$

The set of floating-point classical intervals is denoted by $\updownarrow \overline{R}$.

The floating-point CI arithmetic is obtained by applying the $\updownarrow$ operator for each operation $\circ \in \{\overline{+}, \overline{-}, \overline{\times}, \overline{\div}\}$. For $\updownarrow \overline{x}, \updownarrow \overline{y} \in \updownarrow \overline{R}$, we define $\updownarrow \overline{x} \circ \updownarrow \overline{y} = \updownarrow (\updownarrow \overline{x} \circ \updownarrow \overline{y})$.

Note that, since $\downarrow x \leqslant x \leqslant \uparrow x$, $\overline{x} \subseteq \updownarrow \overline{x}$, and the extended $\circ$ gives an overapproximation, i.e., $\overline{x} \circ \overline{y} \subseteq \updownarrow (\updownarrow \overline{x} \circ \updownarrow \overline{y})$.

This is confirmed by two steps:

- For $\overline{x}, \overline{y} \in \overline{R}$, $\overline{x} \subseteq \updownarrow \overline{x}$ and $\overline{y} \subseteq \updownarrow \overline{y}$. Hence, for $\circ \in \{+, -, \times, \div\}$, $(\overline{x} \circ \overline{y}) \subseteq (\updownarrow \overline{x} \circ \updownarrow \overline{y})$.
- By definition, $(\updownarrow \overline{x} \circ \updownarrow \overline{y}) \subseteq \updownarrow (\updownarrow \overline{x} \circ \updownarrow \overline{y})$.

Floating-point affine interval

For AI, instead of truncating coefficients in an appropriate way, we simply introduce a new noise symcol.

Definition 16 An **Floating-point affine interval** of AI $\ddot{x} = x_0 + \sum_{k=1}^{n} x_k \varepsilon_k \in \ddot{R}$ is a formula

$$\updownarrow \ddot{x} = \downarrow x_0 + \sum_{k=1}^{n} \downarrow x_k \varepsilon_k + B\varepsilon_{n+1}$$

where new noise symbol ε_{n+1} is introduced for REs and $B = \sum_{k=0}^{n} (\uparrow x_k - \downarrow x_k)$. The set of floating-point extended affine intervals is denoted by $\updownarrow \ddot{R}$.

The floating-point AI arithmetic is obtained by introducing a new noise symbol for each operation of AI arithmetic. For example, let $\updownarrow \ddot{x} = \downarrow x_0 + \sum_{k=1}^{n} \downarrow x_k \varepsilon_k, \updownarrow \ddot{y} = \downarrow y_0 + \sum_{k=1}^{n} \downarrow y_k \varepsilon_k$ be two floating-point AI. The addition is

$$\updownarrow \ddot{x} + \updownarrow \ddot{y} = \downarrow (\downarrow x_0 + \downarrow y_0) + \sum_{k=1}^{n} \downarrow (\downarrow x_k + \downarrow y_k)\varepsilon_k + B\varepsilon_{n+1}$$

where $B = \sum_{k+1}^{n} (\uparrow (\downarrow x_k + \downarrow y_k) - \downarrow (\downarrow x_k + \downarrow y_k))$.

Floating-point extended affine interval

For EAI, we safely approximate CI coefficients by the floating-point CI.

Definition 17 A **floating-point extended affine interval** of EAI $\widehat{x} = \overline{x}_0 \,\overline{+}\, \sum_{k=1}^{n} \overline{x}_k \varepsilon_k \in \widehat{R}$ is

$$\updownarrow \widehat{x} = \updownarrow \overline{x}_0 \,\overline{+}\, \sum_{k=1}^{n} \updownarrow \overline{x}_k \varepsilon_k.$$

The set of floating-point extended affine intervals is denoted by $\updownarrow \widehat{R}$.

The floating-point EAI arithmetic is obtained by replacing each CI at a coefficeint by the floating-point CI.

From now on, we will apply floating-point CI, floating-point AI, and floating-point EAI, instead of CI, AI, and EAI, respectively.

4 ORE Analysis as Weighted Model Checking

It has been suggested intimate connections between dataflow analysis and model checking [41, 50]. A program is firstly encoded into a model (*transition system*) by abstraction, and a program analysis is formulated as a model checking problem. This is nicely adopted for control flow analysis and/or classical dataflow analysis in Dragon book [1, 28]. However, as natural requests, we intend more richer dataflow, such as quantity properties with more precise treatments on conditional branches. For instance, linear constraint propagation [41], affine relation analysis [49], or ORE constraint analysis [44] are such examples. In these cases, the direct encoding will be a-transition-as-an-environment-transformer, which requires all possible environments as states. This will lead the state explosion problem in model checking.

In 2003, Rep [48] proposed *weighted pushdown model checking*, in which each transition is associated with a weight. A weight directly represents dataflow, that is, how an abstract environment will be transformed, without generating explicit environments as states. This will not improve complexity in theory, but in practice we can combine with an on-the-fly generation of weights, which drastically reduces the search space during model checking.

We follow this weighted model checking approach (but without using a pushdown stack). Due to infinity (or unboundedness) of weight domains for the ORE analysis, we restrict ourselves to acyclic models only. This is a strong limitation, but our main application target is DSP algorithms, which typically consists of loops with bounded number of iterations and arrays with fixed lengths. An effective widening operation design to avoid this restriction is left for future work. We also put an input range for the ORE analysis, since a narrower input range results more precise analysis results. As side effect, this also enables us to generate weights in an on-the-fly manner.

4.1 Dataflow Analysis as Weighted Model Checking

4.1.1 Weighted Model Checking

Weighted model checking computes dataflow (or, an update of environments) by associating a *weight* to each transition in the model, and the goal is to determine the weight summary of the meet-over-all-path.

Weight domain and weighted transition system. In weighted model checking, the weight domain D is an idempotent semiring.

Definition 18 An **idempotent semiring** is a quintuple $(D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, where $\mathbf{0}, \mathbf{1} \in D$ and $\oplus$, $\otimes$ are binary operators on D such that, for $a, b, c \in D$,

- $(D, \oplus)$ is a commutative monoid with the unit $\mathbf{0}$,
- $(D, \otimes)$ is a monoid with the unit $\mathbf{1}$,
- $\otimes$ distributes over $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$,
- $\oplus$ is idempotent, i.e., $a \oplus a = a$, and
- $\overline{0}$ is the zero element of $\otimes$, i.e., $a \otimes \mathbf{0} = \mathbf{0} \otimes a = \mathbf{0}$.

In the context of dataflow analysis, each element of an idempotent semiring is regarded as follows:

- $\mathbf{0}$ stands for interruption of dataflow,
- $\mathbf{1}$ stands for the identity function (i.e., no state update),
- $\otimes$ is the composition of two successive dataflow, and
- $\oplus$ merges two dataflow at the meet of two transition sequences.

The *weighted transition system* is then defined as *a transition system "plus" a weight domain.*

Definition 19 Let $\mathcal{P} = (P, \Delta, s_0)$ be a transition system with P to be a finite set of states, $\Delta (\subseteq P \times P)$ to be a set of transitions, and $s_0 (\in P)$ to be an initial state. A **weighted transition system** (WTS) is a triplet $\mathcal{W} = (\mathcal{P}, S, f)$, where $\mathcal{S} = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is an idempotent semiring and $f : \Delta \to D$ is a map that assigns a weight to each transition.

Let Δ^* be the set of all **sequences of transitions**. For $\sigma = [r_1, \ldots, r_k] \in \Delta^*$, we define $v(\sigma) =_\Delta f(r_1) \otimes \ldots \otimes f(r_k)$. If σ is a transition sequence from a state c to a state c', we denote $c \Rightarrow^\sigma c'$. The set of all such sequences is denoted by $paths(c, c')$, i.e.,

$$paths(c, c') = \{\sigma \mid c \Rightarrow^\sigma c'\}$$

Weighted model checking. Weighted model checking finds the weight summary of $paths(c, c')$, which is the summation $\oplus_{\sigma \in paths(c,c')} v(\sigma)$.

There are two kinds of generalized reachability problems:

Definition 20 Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted transition system with $\mathcal{P} = (P, \Delta, s_0)$. Let $C \subseteq P$ and $c \in P$.

- The **generalized predecessor problem** is to find $\delta(c) = \oplus\{v(\sigma) \mid \sigma \in path(c, c'), c' \in C\}$.
- The **generalized successor problem** is to find $\delta(c) = \oplus\{v(\sigma) \mid \sigma \in path(c', c), c' \in C\}$

If a cycle exists in a weighted model, $paths(c, c')$ becomes infinite. For the termination of a weighted model checking, an idempotent semiring needs to be *bounded*.

Definition 21 An idempotent semiring is **bounded** if there are no infinite descending chains wrt $\sqsubseteq$, where $a \sqsubseteq b$ if, and only if, $a \oplus b = a$.

4.2 Weight Domain for ORE Analysis

For an ORE problem, we abstract a concrete environment as an abstract environment by using intervals.

Definition 22 Let Var be the set of all variables of the program. An **abstract environment** at a program location is the set of functions $AbsEnv = \{Var \to \Phi_\perp^k\}$, where $k = |Var|$ and $\Phi_\perp \in \{\overline{\Phi}_\perp, \ddot{\Phi}_\perp, \hat{\Phi}_\perp\}$. We define the *zero environment* $e_{\mathbf{0}} \in AbsEnv$ by $e_{\mathbf{0}}(x) = \perp$ for $x \in Var$. Let $e, e' \in AbsEnv$, and **environment meet operation** is defined below:

$$e \sqcup e' = \lambda x.e(x) \sqcup e'(x)$$

where $\sqcup \in \{\overline{\sqcup}, \ddot{\sqcup}, \hat{\sqcup}\}$.

Weight design

The standard definition of a weight domain has the base set of weights $D = AbsEnv \to AbsEnv$. We then theoretically define the weight domain for D as follows:

Definition 23 The weight domain (bounded idempotent semiring) $\mathcal{S} = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ with

$$D = AbsEnv \to AbsEnv,$$
$$\mathbf{1} = \lambda x.x,$$
$$\mathbf{0} = \lambda x.e_{\mathbf{0}},$$
$$w_1 \oplus w_2 = \begin{cases} \lambda x.w_1(x) \sqcup w_2(x) & \text{if } w_1, w_2 \neq \mathbf{0} \\ w_1 & \text{if } w_2 = \mathbf{0} \\ w_2 & \text{if } w_1 = \mathbf{0} \end{cases}$$
$$w_1 \otimes w_2 = \begin{cases} w_2 \cdot w_1 & \text{if } w_1, w_2 \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

where $\sqcup \in \{\overline{\sqcup}, \ddot{\sqcup}, \widehat{\sqcup}\}$.

However, this does not satisfy the descending chain condition (boundedness), since intervals are infinitely many (thus the abstract domain is infinite). To cope with this problem, we:

- restrict the models to be acyclic,
- fix an initial abstract environment I, and
- generate weight on-the-fly.

In the context of our ORE analysis, the intuition behind the first two is,

- a target program has bounded loops only; thus after unfolding loops, abstraction produces an acyclic transition system, and
- the result of ORE analysis depends heavily on the input value; we will set a possible range of inputs at the program entry in advance.

On-the-fly weight generation

We first introduce the augmented weight domain to associate an input abstract environment to each weight. "_" means any input.

Definition 24 The *augmented weight domain* $\mathcal{S}^+ = (D^+, \oplus, \otimes, \mathbf{0}^+, \mathbf{1}^+)$ consists of $D^+ = \{(W, w) \mid W \in AbsEnv, w \in D\}$, $\mathbf{0}^+ = (\_, \mathbf{0})$, $\mathbf{1}^+ = (\_, \mathbf{1})$, and

$$w_1^+ \oplus w_2^+ = \begin{cases} (W_1, w_1 \oplus w_2) & \text{if } W_1 = W_2 \\ \mathbf{0}^+ & \text{otherwise} \end{cases}$$
$$w_1^+ \otimes w_2^+ = \begin{cases} (W_2, w_1 \otimes w_2) & \text{if } W_1 = w_2(W_2) \\ \mathbf{0}^+ & \text{otherwise} \end{cases}$$

for $w_1^+ = (W_1, w_1), w_2^+ = (W_2, w_2) \in D^+$.

Now we are ready to define the on-the-fly weight domain $\mathcal{S}^+_{\mathcal{P},I}$ for a transition system $\mathcal{P}$ and $I \in AbsEnv$. The intuition is, starting from the initial abstract environment I, only reachable instances of weights are computed in on-the-fly manner.

Definition 25 For a transition system $\mathcal{P}$ and $I \in AbsEnv$, the weight domain $\mathcal{S}^+_{\mathcal{P},I} = (D^+_{\mathcal{P},I}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a sub semiring of $\mathcal{S}^\times$ with $D^+_{\mathcal{P},I} \subseteq D^+$. $D^+_{\mathcal{P},I}$ is given by

$$\left\{ (W, w) \;\middle|\; \begin{array}{l} \exists \sigma, \sigma' \in \Delta^* \; \exists c, c' \in P.\; s_0 \Rightarrow^\sigma c \Rightarrow^{\sigma'} c' \\ \wedge W = v(\sigma)(I) \wedge w = v(\sigma') \end{array} \right\}$$

In implementation, we will identify $D^+ \subseteq AbsEnv \times D$ with $D^+ \subseteq AbsEnv \times AbsEnv$ by

$$(W, w) \equiv (W, w(W))$$

for $W \in AbsEnv, w \in D$.

4.3 ORE Analysis

The ORE analysis problem will be solved as weighted model checking on acyclic models by the following steps (Fig. 5).

1. As preprocessing, translate a C program into CIL (three address code language). Then, each loop are unfolded and each array is replaced with a set of variables (as many as its length). We obtain an acyclic program without arrays.
2. Generate weighted transition system, which is a control flow graph with an associated ORE arithmetics operation corresponding to a CIL instruction. ORE arithmetics is prepared for three types (CI, AI, EAI).
3. Apply weighted model checking. During model checking, weights are generated by an on-the-fly manner from given initial ranges of input parameters.

4.3.1 Abstract domain for ORE problem

Abstract domain

The abstract value of a variable aims to cover all of its possible values at one program location. For the ORE problem, the abstract value is a pair of fixed point and roundoff error ranges. We will show three kinds of abstractions based on CI, AI, and EAI range representations.

Definition 26 Let fxp and rdf be corresponding range representations of fixed point and roundoff error.
CI abstract domain $\overline{\Phi} = \{(fxp, rdf) | fxp, rdf \in \overline{R}\}$
AI abstract domain $\ddot{\Phi} = \{(fxp, rdf) | fxp, rdf \in \ddot{R}\}$
EAI abstract domain $\widehat{\Phi} = \{(fxp, rdf) | fxp, rdf \in \widehat{R}\}$

For a fresh symbol $\perp$ (which stands for *undefined* or *uninitialized*), we define $\Phi_\perp = \Phi \cup \{\perp\}$.

Abstract arithmetic

Abstract arithmetic aims to propagate both fixed point ranges and roundoff error ranges of variables.

Definition 27 Replacing (x_f, x_r), (y_f, y_r), $\circledast$, and ϵ in the definition of ORE arithmetic (Definition 8) with

- $(\overline{x}_f, \overline{x}_r)$, $(\overline{y}_f, \overline{y}_r)$, $\overline{\circledast} = \{\overline{\boxplus}, \overline{\boxminus}, \overline{\boxtimes}, \overline{\boxdiv}\}$, and $\overline{\epsilon}$, we obtain **CI abstract arithmetic**,
- $(\ddot{x}_f, \ddot{x}_r)$, $(\ddot{y}_f, \ddot{y}_r)$, $\ddot{\circledast} = \{\ddot{\boxplus}, \ddot{\boxminus}, \ddot{\boxtimes}, \ddot{\boxdiv}\}$, and $\ddot{\epsilon}$, we obtain **AI abstract arithmetic**, and
- $(\widehat{x}_f, \widehat{x}_r)$, $(\widehat{y}_f, \widehat{y}_r)$, $\widehat{\circledast} = \{\widehat{\boxplus}, \widehat{\boxminus}, \widehat{\boxtimes}, \widehat{\boxdiv}\}$, and $\widehat{\epsilon}$, we obtain **EAI abstract arithmetic**,

where

$$\begin{cases} \overline{\epsilon} = \widehat{\epsilon} = [b^{-fp}/2, b^{-fp}/2] \\ \ddot{\epsilon} = (b^{-fp}/2)\varepsilon_r \text{ with a fresh noise symbol } \varepsilon_r \end{cases}$$

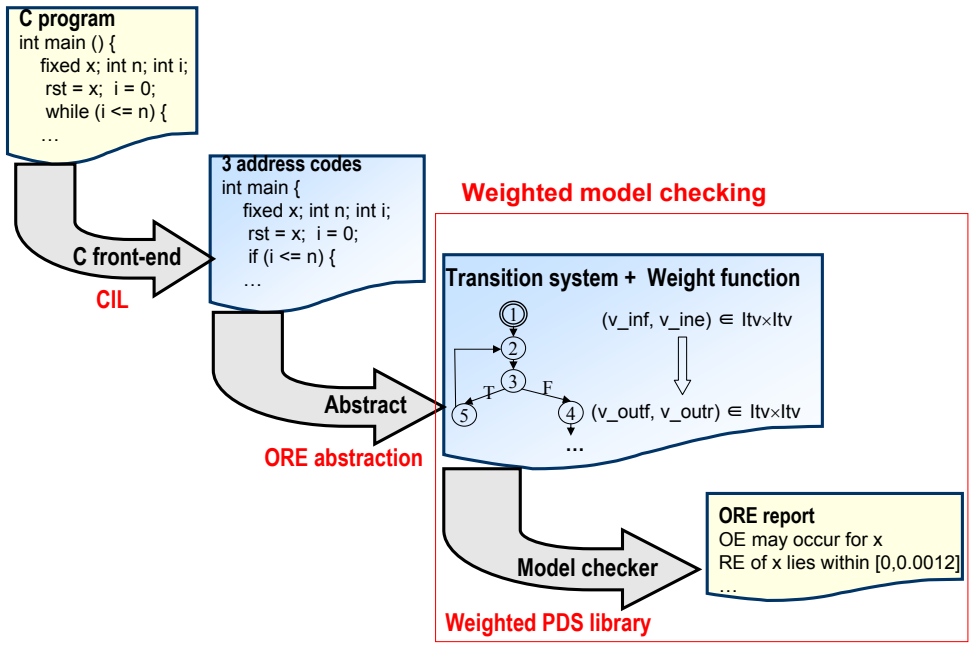


Fig. 5 ORE analysis as weighted model checking

To illustrate how to create EAI abstract numbers and compute EAI abstract arithmetic, we consider the following example:

Example 15 The EAI abstract numbers $(\hat{x}_f, \hat{x}_r)$, $(\hat{y}_f, \hat{y}_r)$ for variables x, y, respectively, in Example 1 are

$$\begin{cases} \hat{x}_f = [1,\ 1] \mp [2,\ 2]\varepsilon_1 \\ \hat{x}_r = [2^{-5},\ 2^{-5}]\varepsilon_3 \end{cases} \quad \begin{cases} \hat{y}_f = [0,\ 0] \mp [10,\ 10]\varepsilon_2 \\ \hat{y}_r = [2^{-5},\ 2^{-5}]\varepsilon_4 \end{cases}$$

since $fp = 5$, the REs of x and y are in $[-2^{-fp-1}, 2^{-fp-1}] = [-2^{-5}, 2^{-5}]$ $(= [-0.03125, 0.03125])$ and $\delta_\times = 2^{-6} - 2^{-11} + 2^{-43} \approx 2^{-6}$.

Then, at the line 2 $(rst = x * x)$, $(\widehat{\texttt{rst}}_f, \widehat{\texttt{rst}}_r) = (\hat{x}_f, \hat{x}_r)\ \hat{\boxtimes}\ (\hat{x}_f, \hat{x}_r)$ *is projected as follows:*

$$\begin{aligned} \widehat{\texttt{rst}}_f &= \hat{x}_f \mathbin{\hat{\times}} \hat{x}_f \\ &= ([1,1] \mp [2,2]\varepsilon_1) \mathbin{\hat{\times}} ([1,1] \mp [2,2]\varepsilon_1) \\ &= [1,1] \mp [0,8]\varepsilon_1 \end{aligned}$$

$$\begin{aligned} \widehat{\texttt{rst}}_r &= 2\hat{\times}\hat{x}_f\hat{\times}\hat{x}_r \mathbin{\hat{+}} \hat{x}_r\hat{\times}\hat{x}_r\hat{+}\delta_\times \\ &= [-0.031250, 0.031250] \mp [-0.123091, 0.123091]\varepsilon_1 \\ &\quad \mp [0.059615, 0.065385]\varepsilon_2 \end{aligned}$$

Abstract comparison operations

Instead of nondeterministic transitions at a conditional branch, the conditional expression can often be evaluated by using abstract environment. This is useful in avoiding unnecessary execution paths. The abstract comparison operations are defined by using ORE comparisons as follows:

Definition 28 Replacing (x_f, x_r), (y_f, y_r), in the definition of $\tilde{x}$ in ORE comparisons (Definition 6) with

- $(\overline{x}_f, \overline{x}_r)$, $(\overline{y}_f, \overline{y}_r)$, we obtain **CI abstract comparison operations** $\hat{\boxtriangle} = \{\hat{\leqslant}, \hat{=}\}$,
- $(\ddot{x}_f, \ddot{x}_r)$, $(\ddot{y}_f, \ddot{y}_r)$, we obtain **AI abstract comparison operations** $\ddot{\boxtriangle} = \{\ddot{\leqslant}, \ddot{=}\}$, and
- $(\hat{x}_f, \hat{x}_r)$, $(\hat{y}_f, \hat{y}_r)$, we obtain **EAI abstract comparison operations** $\hat{\boxtriangle} = \{\hat{\leqslant}, \hat{=}\}$.

The following example illustrates how to evaluate EAI abstract comparison $\hat{<}$.

Example 16 Use $(\hat{x}_f, \hat{x}_r)$, and $(\hat{y}_f, \hat{y}_r)$ as in Example 17. $(\hat{x}_f, \hat{x}_r) \mathbin{\hat{<}} 0$ is evaluated as follows:

- $\overline{x}_f = [-1, 3]$
- $\overline{x}_r = [-2^{-5}, 2^{-5}$

Since $(\overline{x}_f, \overline{x}_r) < 0$ is **unknown**, we can conclude that $(\hat{x}_f, \hat{x}_r) \mathbin{\hat{<}} (\hat{y}_f, \hat{y}_r)$ is **unknown**.

Meet operation

At the meet of two paths in a program, we need to combine the results that are generated from these paths. The result of the meet must bind all input abstract values. We first consider how to compute the union of two ranges:

Definition 29 The unions of ranges are:

- *CI:* $[x_l, x_h] \overline{\cup} [y_l, y_h] = [min(x_l, y_l), max(x_h, y_h)]$.

– *AI:* $(u_0+\sum_{i=1}^{n} u_i\varepsilon_i) \ddot{\cup} (v_0+\sum_{i=1}^{n} v_i\varepsilon_i) = (\frac{u_0+v_0}{2} + \frac{|u_0-v_0|}{2}\varepsilon_{n+1} + \sum_{i=1}^{n} t_i\varepsilon_i)$ where $\varepsilon_{n+1} \in [-1,1]$ is a new noise symbol and, for each i,

$$t_i = \begin{cases} u_i & \text{if } |u_i| > |v_i|, \\ v_i & \text{otherwise.} \end{cases}$$

– *EAI:* $(\overline{u}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{u}_i\varepsilon_i) \mathbin{\hat{\cup}} (\overline{v}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{v}_i\varepsilon_i) = (\overline{u}_0 \mathbin{\overline{\cup}} \overline{v}_0) \mathbin{\overline{+}} \sum_{i=1}^{n} (\overline{u}_i \mathbin{\overline{\cup}} \overline{v}_i)\varepsilon_i$.

Then, the result of meet operation is a pair of the union of fixed point ranges and the union of roundoff error ranges.

Definition 30 The meets in abstract values are:

– *CI meet:* $(\overline{x}_f, \overline{x}_r) \sqcup (\overline{y}_f, \overline{y}_r) = (\overline{x}_f \mathbin{\overline{\cup}} \overline{y}_f, \overline{x}_r \mathbin{\overline{\cup}} \overline{y}_r)$
– *AI meet:* $(\ddot{x}_f, \ddot{x}_r) \ddot{\sqcup} (\ddot{y}_f, \ddot{y}_r) = (\ddot{x}_f \ddot{\cup} \ddot{y}_f,\ \ddot{x}_r \ddot{\cup} \ddot{y}_r)$
– *EAI meet:* $(\hat{x}_f, \hat{z}_r) \hat{\sqcup} (\hat{y}_f, \hat{y}_r) = (\hat{x}_f \hat{\cup} \hat{y}_f,\ \hat{x}_r \hat{\cup} \hat{y}_r)$

$\sqcup \in \{\sqcup, \ddot{\sqcup}, \hat{\sqcup}\}$ is extended to $\Phi_\perp \in \{\overline{\Phi}_\perp, \ddot{\Phi}_\perp, \hat{\Phi}_\perp\}$ by $\perp \sqcup (x_f, x_r) = (x_f, x_r) \sqcup \perp = (x_f, x_r)$.

4.3.2 Weighted transition system for ORE analysis

In preprocessing phase, C programs are transformed into CIL, a three address code language. We replace each array with fixed length by a set of variables that correspond to locations in an array. Thus, our target instructions are restricted as follows.

– **Assignment:** "$x = y \circ z$" with $\circ \in \{+, -, *, /\}$.
– **Conditional instruction:** "if $x \circ y$ then s" where s is an instruction and $\circ \in \{<, <=, >, >=, =, ! =\}$. If the condition $(x \circ y)$ is false, s is not visited; otherwise, s is visited.
– **Control instruction:** "return *loc*", "goto *loc*", "break", "continue". Control moves to the specified location, and the values of variables do not change.
– **While Loop:** "while $x \circ y$ { *body* }" with $\circ \in \{<, <=, >, >=, =, ! =\}$. *body* is repeated as long as the condition $(x \circ y)$ holds. Inside *body*, "break" will exit from the loop.

In preprocessing phase, the bounded while loops are unfolded as a sequence of conditional instructions, and we obtain an *acyclic* CIL program.

The weight function is defined as follows:

Definition 31 For an acyclic transition system $\mathcal{P}$ and $I \in AbsEnv$, the weight function $f_{\mathcal{P},I} : \Delta \to D^+_{\mathcal{P},I}$ is given in Table 2.

Then, we obtain the weighted transition system:

$$\mathcal{W} = (\mathcal{P}, \mathcal{S}^+_{\mathcal{P},I}, f_{\mathcal{P},I}).$$

We explain how to generate a weighted transition system by Example 1 (in Introduction).

instruction	weight
"$x = y \circ z$"	$(W_i, \{x_o = y_i \boxed{\circ} z_i,\ v_o = v_i \mid v \in Var\backslash\{x\}\})$ where $\boxed{\circ}$ is the corresponding abstract arithmetic operation of $\circ$
"if $x \circ y$ then s"	$\mathbf{0}^+$ if $x_i \boxed{\circ} y_i$ = **false**; $\mathbf{1}^+$ otherwise, where $\boxed{\circ}$ is the corresponding abstract comparison of $\circ$
Control inst.	$\mathbf{1}^+$

Table 2 Weight function of ORE analysis

```
maintest{
st1.    if (x > 0) {
st2.      rst = x * x;}
        else {
st3.      __cil_tmp1 = (Real )3;
        rst = __cil_tmp1 * x;}
st4.    rst -= y;
st5.    return (rst);
st6.  }
```

Fig. 6 CIL code for Example 1

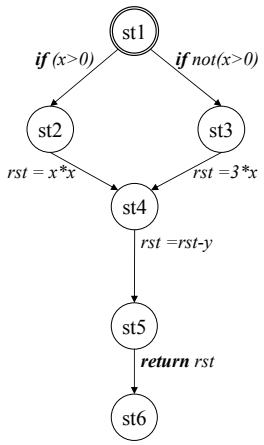


Fig. 7 CFG of three address codes in Fig. 6

Example 17 We use EAI representation type.

The CIL codes of the C program in Fig. 2 are shown in Fig. 6. Let $st1, ..., st6$ be its locations.

To distinguish variables at each locations, we will denote a variable v at the location sti by $v^{(i)}$. The fixed-point value and RE of v are denoted by $\hat{v}_f^{(i)}$, $\hat{v}_r^{(i)}$ respectively.

The transition system is $\mathcal{P} = (P, \Delta)$, where $P = \{st1, st2..., st6\}$, Δ is shown in Fig. 7 and f is defined in Table 3.

The initial abstract environment W_{init} at $st1$ is generated from initial range values of variables (given in the topmost comments in Fig. 2), in that:

$$\begin{cases} x^{(init)} = ([1,1] \mathbin{\overline{+}} [2,2]\varepsilon_1,\ [2^{-5}, 2^{-5}]\varepsilon_3) \\ y^{(init)} = ([10,10]\varepsilon_2,\ [2^{-5}, 2^{-5}]\varepsilon_4) \end{cases}$$

Then, the resulting weighted transition system is $\mathcal{W} = (\mathcal{P}, \mathcal{S}^{+}_{\mathcal{P},W_{init}}, f)$.

transition	weight
(st1,st2)	$if\ ((x^{(init)} \gtrless 0) = false)\ \mathbf{0}^+\ else\ \mathbf{1}^+$
(st1,st3)	$if\ not((x^{(init)} \gtrless 0) = false)\ \mathbf{0}^+\ else\ \mathbf{1}^+$
(st2,st4)	$(W_i^{(in)},\ \{rst^{(2)} = x^{(in)}\ \hat{\boxtimes}\ x^{(in)},$ $v^{(2)} = v^{(in)} \mid v \in Var\backslash\{rst\}\})$
(st3,st4)	$(W^{(in)},\ \{\_\_cil\_tmp1^{(3)} = 3,$ $rst^{(3)} = \_\_cil\_tmp1^{(3)}\ \hat{\boxtimes}\ x^{(in)},$ $v^{(3)} = v^{(in)} \mid v \in Var\backslash\{\_\_cil\_tmp1, rst\}\})$
(st4,st5)	$(W^{(in)},\ \{rst^{(4)} = rst^{(in)}\ \hat{\boxminus}\ y^{(in)},$ $v^{(4)} = v^{(in)} \mid v \in Var\backslash\{rst\}\})$
(st5,st6)	$\mathbf{1}^+$

Table 3 Weight function for a CIL code in Example 17

Since the abstraction is an overapproximation, we conclude soundness of ORE analysis.

Theorem 2 *An ORE analysis on a subclass of acyclic CIL programs is sound.*

4.3.3 ORE Analysis Example

We continue to explain how ORE analysis works by the example in Fig. 2.

Example 18 Let the input EAI $x^{(1)} = (\hat{x}_f^{(1)}, \hat{x}_r^{(1)})$ and $y^{(1)} = (\hat{y}_f^{(1)}, \hat{y}_r^{(1)})$ as shown in Example 17.

$St1$ is a conditional branch. Since the initial range of x is $[-1, 3]$, CANA cannot decide the condition $x > 0$. Thus, it traces both st2 and st3, and later merges their results.

At $st2$, the RE of `rst` is computed by multiplication $\hat{\boxtimes}$ as:

$$\widehat{\texttt{rst}}_r^{(2)} = [-0.031250, 0.031250] \overline{+} [-0.123091, 0.123091]\varepsilon_1 \overline{+} [0.059615, 0.065385]\varepsilon_3$$

and at st3,

$$\widehat{\texttt{rst}}_r^{(3)} = 3 \hat{\times} \hat{x}_r = [0.093750, 0.093750]\varepsilon_3$$

And they are merged as:

$$\begin{aligned}\widehat{\texttt{rst}}_r &= [-0.031250, 0.031250] \overline{+} [-0.123091, 0.123091]\varepsilon_1 \overline{+} ([0.059615, 0.065385] \cup [0.093750, 0.093750])\varepsilon_3 \\ &= [-0.031250, 0.031250] \overline{+} [-0.123091, 0.123091]\varepsilon_1 \overline{+} ([0.059615, 0.093750])\varepsilon_3\end{aligned}$$

At $st4$, by subtraction $\hat{\boxminus}$, we get the RE of `rst`:

$$\widehat{rst}_r^{(4)} = [-0.031250, 0.031250] \overline{+} [-0.123091, 0.123091]\varepsilon_1 \overline{+} [0.059615, 0.09375]\varepsilon_3 \overline{+} [-0.031250, -0.031250]\varepsilon_4$$

RE bound of `rst` is by $[-0.279341, 0.279341]$ by replacing each ε_i with $[-1, 1]$ in $\hat{r}$.

By nature of analysis, this analysis overapproximates REs. It occurs at the conditional branch (line 1) and the multiplication. For instance, at $st2$, $\hat{\delta}$ (in $\hat{\boxtimes}$) is approximated with $[-0.031250, 0.031250]$.

The example above shows that the overapproximations will occur at (1) nonlinear operations; (2) undecided conditional branch. In case the conditional branch is decided, analysis will not overapproximate it. Below, how the treatment of conditional branches in ORE analysis will improve the overapproximation.

Example 19 In Example 18, if we reduce the initial range of x_f to $[1, 3]$, the condition $x > 0$ is decided to be true at st1, and CANA ensures that $st3$ will not be executed. Then, at $st4$, by substraction $\hat{\boxminus}$,

$$\hat{r} = [-0.031250,\ 0.031250] \overline{+} [-0.123091,\ 0.123091]\varepsilon_1 \overline{+} [0.059615, 0.065385]\varepsilon_3 \overline{+} [-0.031250,\ -0.031250]\varepsilon_4$$

RE of `rst` is bounded by $\overline{r} = [-0.250976, 0.250976]$ by replacing each ε_i with $[-1, 1]$ in $\hat{r}$.

This result is more precise than that in Example 18.

4.4 Experiments

Implementation

We have implemented our analysis framework in a tool *C ANAlyzer* (CANA). CANA uses two libraries: CIL library [3] and WPDS library [4].

- CIL (C Intermediate Language) is a high-level representation that permit source-to-source transformation of C programs. CIL is used to generate three address codes, information about variables, and the CFG of a C program.
- WPDS (Weighted Pushdown System) is a library, which provides functions to the sets of forward- or backward- reachable configurations in a weighted pushdown system. Since we exclude procedure calls and unbounded loops, we adopt WPDS only for weighted finite acyclic state transition systems

The inputs of CANA are subclass of ANSI C programs and initial ranges of variables. The outputs of CANA are roundoff error ranges of variables at each point of the program, and warning about overflow errors (if they occur). CANA has six main modules (Fig. 5.4) as follows:

[3] http://hal.cs.berkeley.edu/cil/
[4] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/

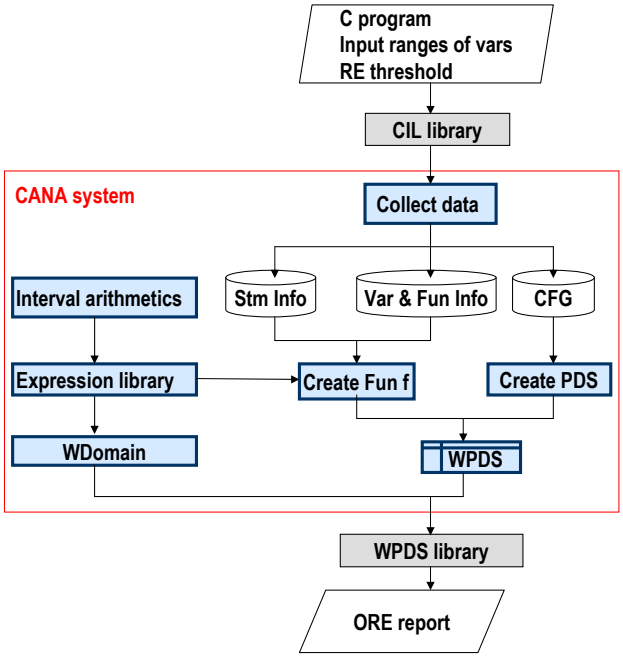


Fig. 8 CANA system

1. *Collect data* module generates information required for the analysis, including: statement information (Stm Info), (2) variable and function information (Var and Func Info), and (3) CFG of C program.
2. *Range arithmetics* module includes three types of interval arithmetics: CI arithmetic, AI arithmetic, and EAI arithmetic.
3. *Evaluate exps* module evaluates the abstract values of expressions based on types of range arithmetics.
4. *Create PDS* module generates transition system from control flow graph of a C program.
5. *Create Fun f* module assigns a weight to each transition.
6. *WDomain* module defines two operations: $\otimes$ and $\oplus$.

Experimental results

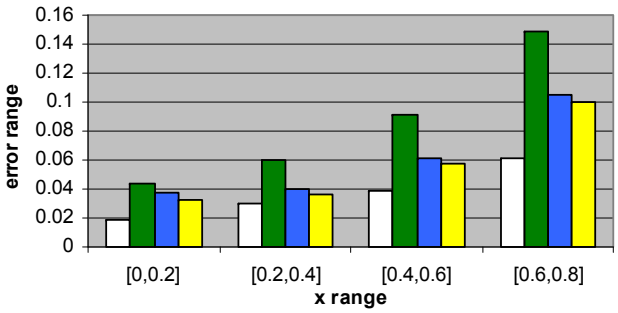
We have implemented in CANA three types of interval representations: CI, AI, and EAI. CANA can analyze programs that have nested loops 64×64.

In order to compare the efficiency of EAI arithmetic to CI and AI arithmetics, we analyzed source codes of three examples:

1. a program computes a polynomial of degree 5
2. a program computes the sine function
3. a tiny fragment, which frequently appears in the mpeg decoder reference algorithm.

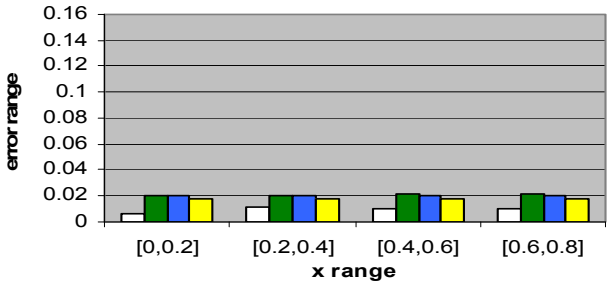
Fig. 9 shows experimental results of analyzing these programs (on PC with Intel(R) Core(TM) Duo CPU

[x]	[0,0.2]	[0.2,0.4]	[0.4,0.6]	[0.6,0.8]
real err	0.01909	0.03	0.03891	0.061588
CI	0.04373	0.0599	0.09172	0.148848
AI	0.0377	0.0406	0.06179	0.104468
EAI	0.03232	0.0356	0.05763	0.100454



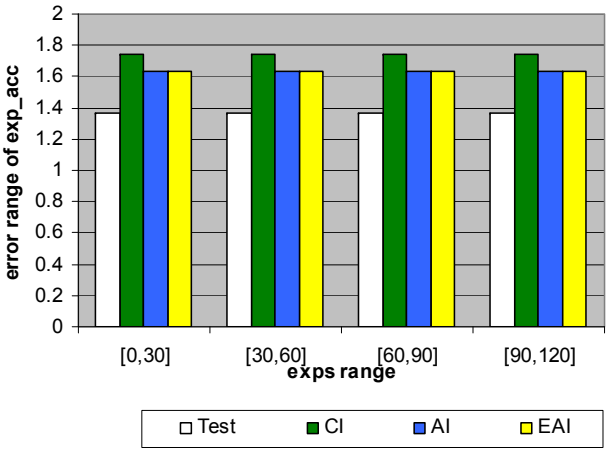
a. The analysis result of P5(x)

[x]	[0,0.2]	[0.2,0.4]	[0.4,0.6]	[0.6,0.8]
real err	0.00647	0.0108	0.01049	0.010206
CI	0.0204	0.0208	0.02139	0.022198
AI	0.02035	0.0204	0.02029	0.019976
EAI	0.01759	0.0176	0.01749	0.017184



b. The analysis result of Sin(x)

[exps]	[0,30]	[30,60]	[60,90]	[90,120]
real err	1.36309	1.3631	1.36309	1.363086
CI	1.73766	1.7377	1.73766	1.737656
AI	1.63453	1.6345	1.63453	1.634531
EAI	1.63453	1.6345	1.63453	1.634531



c. The analysis result of pMpeg(exps)

Fig. 9 The experimental results

1.66GHz, 1.5Gb of memory). Fig. 9a shows results of analyzing the program that computes $P5(x) = 1 - x + 3x^2 - 2x^3 + x^4 - 5x^5$, where the fraction part $fp = 8$. The *true err* row is the width of roundoff error ranges which are found by testing; the CI, AI, and EAI rows are the widths of roundoff error ranges computed by CI arithmetic, AI arithmetic, and EAI arithmetic, respectively. Our experiments show that EAI is more precise than CI and is comparable to AI. We got similar results for the program that computes sine of x shown in Fig. 9b, and the fragment of the mpeg decoder $pMpeg(exps)$ in Fig. 9c. All tests run in less than 2 seconds.

5 Detecting REs based on Counterexample-guided Narrowing

Static analysis is useful in proving safety properties of ORE problem. But it requires overapproximation in both propagating REs and control flows like conditional branches. Hence, it may return spurious counterexamples. Fortunately, in our setting, the floating to fixed-point conversion, we can compute the exact RE, whereas there are in general no ways to compute exact real numbers. Though testing can return exact REs, it cannot cover all possible inputs. If we are lucky, witness of large REs would be eventually found, yet most of them may be missed. Another challenging problem is how to reduce the number of test cases if the input domain is large and the input parameters are many.

A popular approach to deal with spurious counterexamples is *counterexample-guided abstraction refinement* (CEGAR) [6]. Inspired by CEGAR, we combine testing and RE analysis.

This section proposes an approach for detecting REs of C programs, which combines static analysis and testing, and make them refine each other. We call this combination *counterexample-guided narrowing*. First, we apply an overflow and roundoff errors analysis from our previous work [44] which returns an overapproximation of REs as an Extended affine interval (EAI). Fortunately, an EAI represents the relations between the input value and the RE of the output. These relations can be used to clarify: variables are irrelevant to REs of the results, variables affect the REs the most, and the ranges of inputs are most likely to cause the maximum RE. These observations effectively narrow the focus of test data generation. Second, in case testing does not find a witness of RE violation, the analysis may overapproximate too much. Further, the narrower the input ranges are, the more precise the analysis result will be. Therefore, with a "divide and conquer" refinement strategy, we can check the most suspicious part first.

Throughout the section, we focus only on roundoff errors. We assume that ORE analyzer (CANA) does not detect any overflow errors.

5.1 Observation on RE Analysis

The inputs of an RE analysis consist of

- a C program (with m-input variables) to be analyzed (base $b = 2$),
- a fixed-point format (sp, ip, fp),
- an RE threshold $\theta(> 0)$, and
- a pair of a fixed-point range $[l_i, h_i]$ and an RE range $[l_{m+i}, h_{m+i}]$ with $-2^{-fp-1} < l_{m+i} \leqslant h_{m+i} < 2^{-fp-1}$ for each i-th input variable.

We will fix the last three elements as an environment of the RE analysis. We call the Cartesian product $D = [l_1, h_1] \times \cdots \times [l_{2m}, h_{2m}]$ *an input domain*.

Throughout the RE analysis, all ranges are represented as EAIs with $2m$ noise symbols (where ε_i and ε_{m+i} correspond to noise symbols of the fixed-point part and the RE of values of the i-th input variable). Thus, we coerce input CIs to EAIs by EAI coercion. In the context of the RE analysis, we denote:
Input domain $D = [l_1, h_1] \times \cdots \times [l_{2m}, h_{2m}]$ is $(\hat{v}_1, \ldots, \hat{v}_{2m})$ with $\hat{v}_i = (\frac{l_i + h_i}{2}) \mathbin{\overline{+}} (\frac{h_i - l_i}{2})\varepsilon_i$.

As notational convention, the analysis result is denoted by an EAI:

$$\hat{r} = \overline{r}_0 \mathbin{\overline{+}} \sum_{i=1}^{2m} \overline{r}_i \varepsilon_i$$

The analysis result $\hat{r}$ shows extra information about the effects of inputs on $\hat{r}$, since EAI coercions of input ranges and $\hat{r}$ share common noise symbols. If violations are found (i.e., EAI projection of $\hat{r}$ exceeds $[-\theta, \theta]$), we need to check whether they are spurious by testing. Fortunately, we have useful observations below, which will optimize test data generation and testing.

- **RE bound for each test case:** Assume that the valuation of noise symbols for the test case $t = (t_1, \ldots, t_{2m})$ is $(\lambda_1, \ldots, \lambda_{2m})$, i.e.,

$$\lambda_i = \begin{cases} 0 & \text{if } l_i = h_i \\ \frac{2t_i - (l_i + h_i)}{(h_i - l_i)} & \text{otherwise} \end{cases}$$

  Then, the RE for the input t is bounded by the valuation of $\hat{r}$ with $(\lambda_1, \ldots, \lambda_{2m})$.
- **Irrelevant noise symbol**: If the coefficient of a noise symbol ε_k is $\overline{r}_k = [0, 0]$, the noise symbol ε_k will not affect the RE of result.
- **Sensitivity of noise symbols**: If $|\overline{r}_k| \leqslant |\overline{r}_h|$, the noise symbol ε_k affects $\hat{r}$ more than ε_h.

The following example will demonstrate how they affect the testing phases:

Example 20 In the analysis result of Example 18:

- For the test case $t = (x_f, y_f, x_r, y_r) = (1, 5, 0, 0)$, the valuation of noise symbols is $(\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4) = (0, 0.5, 0, 0)$. The RE bound of t is

$$[-0.031250, 0.031250] \subseteq [-0.26, 0.26]$$

  Therefore, t is not a counterexample.
- Since $\overline{r}_2 = [0, 0]$, ε_2 is an *irrelevant noise symbol*. Hence, v_2 (or fixed-point part of y) does not affect RE of rst.
- We have $|\overline{r}_1| = 0.123091 = max\{|\overline{r}_1|, \ldots, |\overline{r}_4|\}$, thus ε_1 is the most sensitive noise symbol. Hence, v_1 (or fixed-point part of x) affect the RE of rst the most.

The analysis result is helpful to optimize test phase, includes:

- **Pre-evaluate test case:** if RE bound of a test case t lies in the RE threshold bound, we need not execute test for t.
- **Reduce input ranges:** which reduces segments in each input range if corresponding REs are subsumed. Especially, for an irrelevant noise symbol.
- **Choice of the number of ticks**, which takes more ticks in the input ranges of more sensitive noise symbols.

5.2 Refining Test Data Generation

5.2.1 Test Data Generation

For an input domain $D = [l_1, h_1] \times \cdots \times [l_{2m}, h_{2m}]$, a basic strategy of test data generation is to divide the input domain into meshes and select one test case from each mesh.

Definition 32 For an interval $[l, h]$, and $k > 1$, the **k-ticks** of $[l, h]$ starting from c are $\{c, c + \Delta, \cdots, c + (k - 1)\Delta\}$ where $\Delta = \frac{h-l}{k}$ and c is a number lies in $[l, l + \Delta]$.

Remark 3 This c intends a randomly generated flicker; since refinement loops will repeat, we would like to avoid generating the same test cases.

Example 21 Let us consider the C program as Fig. 2.

- For $x_f \in [-1, 3]$, let $k_1 = 10$, then $\Delta_1 = (3 - (-1))/10 = 0.4$. Let $c_1 = -0.8$, we got the set of ticks $X_f = \{-0.8, \ -0.4, \ \cdot, \ 3.8\}$
- For $x_r \in [-2^{-5}, 2^{-5}]$, let $k_2 = 10$, then $\Delta_2 = (2^{-5} - (-2^{-5}))/10 = 0.00625$. Let $c_2 = -0.03$, we got the set of ticks $X_r = \{-0.03, \ -0.02375, \ \cdot, \ 0.0265\}$
- For $y_f \in [-10, 10]$, let $k_3 = 10$, then $\Delta_3 = (10 - (-10))/10 = 2$. Let $c_3 = -9$, we got the set of ticks $Y_f = \{-9, \ -7, \ \cdot, \ 9\}$
- For $y_r \in [-2^{-5}, 2^{-5}]$, let $k_4 = 10$, then $\Delta_4 = (2^{-5} - (-2^{-5}))/10 = 0.00625$. Let $c_4 = -0.028$, we got the set of ticks $Y_r = \{-0.028, \ -0.02175, \ \cdot, \ 0.0285\}$

Hence, the set of test data is $T = X_f \times Y_f \times X_r \times Y_r$. E.g., For test case $t_1 = (-0.8, -9, -0.03, -0.028)$, the input of fixed-point program is $(x, y) = (-0.8, -9)$; the input of floating-point program is $(x, y) = (-0.83, -9.028$.

For an input domain $D = [l_1, h_1] \times \cdots \times [l_{2m}, h_{2m}]$, all combinations of k_i-ticks of $[l_i, h_i]$ for $i \leqslant 2m$ are the set of test data. Then, we execute a program in two ways: with floating-point arithmetic and fixed-point arithmetic. The difference between them is a true RE. However, the number of test data grows with the power of the $2m$-th degree.

5.2.2 Range Reduction

For two ranges $[l_1, h_1], [l_2, h_2]$, we denote $[l_1, h_1] \leqslant [l_2, h_2]$ if $u \leqslant v$ for each $u \in [l_1, h_1]$ and $v \in [l_2, h_2]$ (i.e., $h_1 \leqslant l_2$). Reducing input range is executed based on the observation as the following lemma.

Lemma 2 *Assume* $0 \notin [u, v]$. *Then,*

- $[-v, -u] \leqslant [u, v][-\frac{u}{v}, \frac{u}{v}] \leqslant [u, v]$ *if* $0 < u \leqslant v$
- $[u, v] \leqslant [u, v][-\frac{v}{u}, \frac{v}{u}] \leqslant [-v, u]$ *if* $u \leqslant v < 0$

This lemma means that if $0 \notin [u_i, v_i] = \overline{r}_i$ we can ignore test data with corresponding noise symbol ε_i in $[-\frac{u_i}{v_i}, \frac{u_i}{v_i}]$ (resp. $[-\frac{v_i}{u_i}, \frac{v_i}{u_i}]$) for $0 < u_i \leqslant v_i$ (resp. $u_i \leqslant v_i < 0$). The reason for this is that the true REs for test data with ε_i in $[-\frac{u_i}{v_i}, \frac{u_i}{v_i}]$ (resp. $[-\frac{v_i}{u_i}, \frac{v_i}{u_i}]$) are bounded by the valuations when a noise symbol ε_i is either 1 or -1 whatever a true coefficient has a value in $[u_i, v_i]$.

In a special case $u_i = v_i = 0$, a valuation of ε_i does not matter. Thus, only a valuation with 0 is considered.

From observations above we reduce the input domain D to two input domains D_{max} (which contain an input that causes the maximum RE) and D_{min} (which contain an input that causes the minimum RE) without loosing opportunities to find test cases that cause violation of REs.

Definition 33 For an input domain $D = (\frac{l_1+h_1}{2} + \frac{h_1-l_1}{2}\varepsilon_1) \times \cdots \times (\frac{l_{2m}+h_{2m}}{2} + \frac{h_{2m}-l_{2m}}{2}\varepsilon_{2m})$ and the analysis result

$$\widehat{r} = [u_0, v_0] \mathbin{\overline{+}} \sum_{i=1}^{2m} [u_i, v_i]\varepsilon_i,$$

the subdomain D_{max} of D is

$$(\frac{l_1+h_1}{2}+\frac{h_1-l_1}{2}\overline{\varepsilon}_1)\times\cdots\times(\frac{l_{2m}+h_{2m}}{2}+\frac{h_{2m}-l_{2m}}{2}\overline{\varepsilon}_{2m})$$

where

$$\overline{\varepsilon}_i=\begin{cases}[\frac{|u_i|}{|v_i|},1] & \text{if } 0<u_i\leqslant v_i\\ [-1,-\frac{|v_i|}{|u_i|}] & \text{if } u_i\leqslant v_i<0\\ [0,0] & \text{if } u_i=v_i=0\\ [-1,1] & \text{otherwise}\end{cases}$$

and the subdomain D_{min} is

$$(\frac{l_1+h_1}{2}+\frac{h_1-l_1}{2}\underline{\varepsilon}_1)\times\cdots\times(\frac{l_{2m}+h_{2m}}{2}+\frac{h_{2m}-l_{2m}}{2}\underline{\varepsilon}_{2m})$$

where

$$\underline{\varepsilon}_i=\begin{cases}[-1,-\frac{|u_i|}{|v_i|}] & \text{if } 0<u_i\leqslant v_i\\ [\frac{|v_i|}{|u_i|},1] & \text{if } u_i\leqslant v_i<0\\ [0,0] & \text{if } u_i=v_i=0\\ [-1,1] & \text{otherwise}\end{cases}$$

An EAI $\hat{r}=\overline{r}_0\mathbin{\overline{+}}\sum_{i=1}^{2m}\overline{r}_i\varepsilon_i$ is maximum (resp. minimum) if each of elements $\overline{r}_0$ and $\overline{r}_i\varepsilon_i$ is maximized (resp. minimized). Thus the maximal (resp. minimal) RE will occur among valuations of ε_i in $[u/v,1]$ (resp. $[-1,-u/v]$) if $0<u\leqslant v$, and in $[-1,-v/u]$ (resp. $[v/u,1]$) if $u\leqslant v<0$. Therefore, from Lemma 2, we obtain the next theorem.

Theorem 3 *If there exists a counterexample in D, then there exists a counterexample in $D_{max}\cup D_{min}$.*

Example 22 By observation of analysis result $\hat{r}$ in Example 18, we can find D_{max} of the input domain D is

$$([1,1]\mathbin{\overline{+}}[2,2]\overline{\varepsilon}_1)\times([10,10]\overline{\varepsilon}_2)\times([2^{-5},2^{-5}]\overline{\varepsilon}_3)\times([2^{-5},2^{-5}]\overline{\varepsilon}_4)$$

with $\overline{\varepsilon}_1=[-1,1]$, $\overline{\varepsilon}_2=[0,0]$, $\overline{\varepsilon}_3=[0.63589,1]$, and $\overline{\varepsilon}_4=[-1,-1]$. Hence, D_{max} is projected to

$$[-1,3]\times[0,0]\times[0.019872,0.03125]\times[-0.03125,-0.03125].$$

Hence, we can conclude that the input with $y_f=0$ and $y_r\simeq-0.03125$ will cause the maximum RE.

5.2.3 More Ticks for more Sensitive Noise Symbols

We need a strategy to setting ticks for each initial ranges in input domain. A bad strategy of setting ticks will create several test cases which cause similar REs and all of them lie within RE threshold bound. Testing over these test cases are not needed.

Analysis result $\hat{r}$ shows the effects of noise symbols to the REs. A larger coefficient of a noise symbol causes stronger effect on the RE of the result. For example, the variable corresponding to dominant noise symbol will strongly affect REs, in other words, the changing of this variable causes the changing of RE the most. Therefore, setting more ticks on the initial range of this variables can lead the test cases to various REs. Based on this observation, our basic idea is, in input domains D_{min}, D_{max}, the initial range of variables which are predicated strongly affect REs will be set more ticks than other initial ranges. The strategy of setting number of ticks is then depending on coefficients of noise symbols in analysis result as follows:

Definition 34 Let $\hat{r}=\overline{r}_0\mathbin{\overline{+}}\sum_{i=0}^{2m}\varepsilon_i$ be analysis result of input domain $D'=[u_1,v_1]\times\cdots\times[u_{2m},v_{2m}]$.

For $\sigma>0$, a *tick frequency* t_i (wrt σ) for the input interval $[u_i,v_i]$ is

$$t_i=\begin{cases}\lceil|\frac{2\times\overline{r}_i|}{\sigma}\rceil & \text{if } u_i<v_i\\ 1 & \text{if } u_i=v_i\end{cases}$$

Here, $\lceil x\rceil$ denotes the round up of x.

Example 23 For $\hat{r}$ in Example 18 and D_{max} in Example 22, the tick frequency t_1, t_2, t_3, and t_4 wrt $\sigma=0.01$ (of $\overline{\varepsilon}_1$, $\overline{\varepsilon}_2$, $\overline{\varepsilon}_3$, and $\overline{\varepsilon}_4$, respectively) are,

$$\begin{aligned}t_1&=\lceil\tfrac{2\times0.123091}{0.01}\rceil=25\\ t_2&=1\\ t_3&=\lceil\tfrac{2\times0.09375}{0.01}\rceil=19\\ t_4&=1\end{aligned}$$

Thus, the number of test cases is $25\times1\times19\times1=475$, and the RE found by testing 475 test cases is 0.219720.

5.3 Refinement of Analysis by Narrowing Input Domains

An analysis may report spurious counterexamples. Fortunately, our RE analysis becomes more precise if an input domain becomes narrower. There are two reasons that make input domain decomposition reduces the overapproximations:

- a smaller input domain is more likely to be deterministic on conditional branches, and
- a smaller input domain is more likely makes EAI arithmetic more precise.

Our "divide and conquer" strategy has two phases:

- Reduce an input domain D to D_{max} and D_{min} (Definition 33)
- Divide the input ranges (in D_{max} and D_{min}) of the most sensitive noise symbol ε_k into two ranges.

Definition 35 Let $D_{max} = [l_1, h_1] \times \ldots \times [l_{2m}, h_{2m}]$ be an input domain and ε_k be the most sensitive noise symbol. Then:

- $D^1_{max} = D_{max}|_{\overline{v}_k = [l_k, \frac{l_k + h_k}{2}]}$
- $D^2_{max} = D_{max}|_{\overline{v}_k = [\frac{l_k + h_k}{2}, h_k]}$

where $\overline{v}_k$ is the k-th element of D^1_{max} (D^2_{max}).
For D_{min}, we also have a similar partition strategy.

The next round of the RE analysis will be performed for input domains D^1_{max} and D^2_{max}. Our early experience shows that often one of analysis results of D^1_{max} and D^2_{max} lie in the RE threshold bound. Thus this simple strategy becomes quite effective.

Example 24 From Example 20, the most sensitive noise symbol is ε_1.
Form Example 22, the new input domain D_{max} is

$$[-1, 3] \times [0, 0] \times [0.019872, 0.03125] \times [-0.03125, -0.03125]$$

and ε_1 is the most sensitive symbol (Example 20).

Hence, we will divide the initial range of v_1 ($[-1, 3]$) into two new subranges $[-1, 1]$ and $[1, 3]$ and we get:

$$\begin{aligned} D^1_{max} &= D_{max}|_{\overline{v}_1 = [-1,1]} \\ &= [-1, 1] \times [0, 0] \times [0.019872, 0.03125] \times \\ &\quad [-0.03125, -0.03125] \\ D^2_{max} &= D_{max}|_{\overline{v}_1 = [1,3]} \\ &= [1, 3] \times [0, 0] \times [0.019872, 0.03125] \times \\ &\quad [-0.03125, -0.03125] \end{aligned}$$

RE analyses on two domains D^1_{max} and D^2_{max} report that:

- the REs of all input in D^1_{max} lie in $[-0.22, 0.22]$
- the REs of all input in D^2_{max} lie in $[-0.25, 0.25]$

Hence, we can conclude the REs of all input in D_{max} lie in RE threshold bound $[-0.26, 0.26]$.

Similarly, we get the REs of all input in D_{min} also lie in the RE threshold bound, and we can conclude the program is "safe". Note that, before decomposition, it was $[-0.28, 0.28]$ (Example 18), which exceeds the RE threshold bound (Fig. 10). If we reduce RE threshold bound θ in $0.219720 < \theta < 0.22$, both D^1_{max} and D^2_{max} are not enough. In such a case, we will investigate subdomain which has larger RE found testing first in the later rounds.

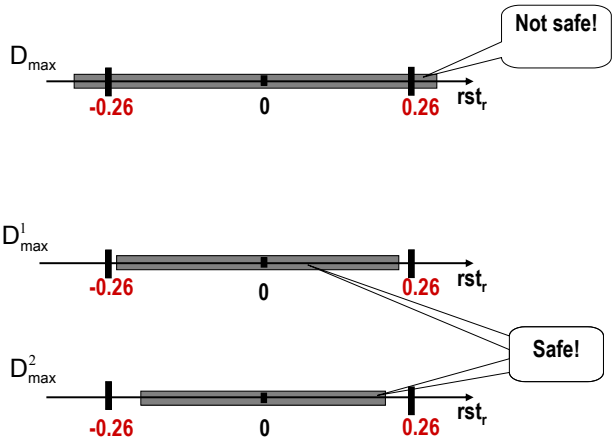


Fig. 10 Effects of decomposition of D_{max} to D^1_{max}, D^2_{max}

Combining Analysis and Testing Algorithm

Algorithm 5.3 shows the algorithm combining the analysis and testing. Function $analyze(P_{fl}, D)$ analyzes the program P_{fl} with input domain D, and return the overapproximate RE in EAI form $\hat{r}$. Function $reduceMax(\hat{r}, D)$ reduce domain D to D_{max}. Function $reduceMin(\hat{r}, D)$ reduce domain D to D_{min}. Function $gentest(D, \hat{r})$ generate a set of test cases T of domain D using analysis result $\hat{r}$. Function $test(P_{fl}, P_{fx}, t)$ executes both two programs P_{fl}, P_{fx} with test case t and return the difference between result of these two program, called set of test results D_R. Function $devide(D, \hat{r})$ divides domain D into two new subdomains D_1, D_2 based on analysis result $\hat{r}$.

Input: P_{fl}, P_{fx}, initial ranges of variables, RE threshold θ
Output: Return "safe" or counterexample or unknown
Initial list of subdomains $L_D = [D]$, a set of test data $T = \varnothing$, a set of RE $D_R = \varnothing$;
while $length(L_D) < 10$ **do**
  pop one element D from L_D;
  $\hat{r} = analyze(P_{fl}, D)$;
  if $(\hat{r} \subseteq [-\theta, \theta])$ **then**
    continue ;
  end
  $D_{max} = reduceMax(\hat{r}, D)$;
  D_{min}=$reduceMin(\hat{r}, D)$;
  push D_{min} into L_D;
  $r_{max} = max\{|test(P_{fl}, P_{fx}, t)||t \in T\}$;
  Let $t_{max} \in T$ such that $r_{max} = test(P_{fl}, P_{fx}, t_{max})$;
  if $(r_{max} > \theta)$ **then**
    return *counterexample* t_{max};
  end
  $\{D_1, D_2\} = divide(D_{max}, \hat{r})$;
  if $t_{max} \in D_1$ **then**
    push D_2 into L_D;
    push D_1 into L_D;
  else
    push D_1 into L_D;
    push D_2 into L_D;
  end
end
return *unknown*;

Algorithm 1: Combining analysis and testing

5.4 Implementation and Experiments

CANAT implementation

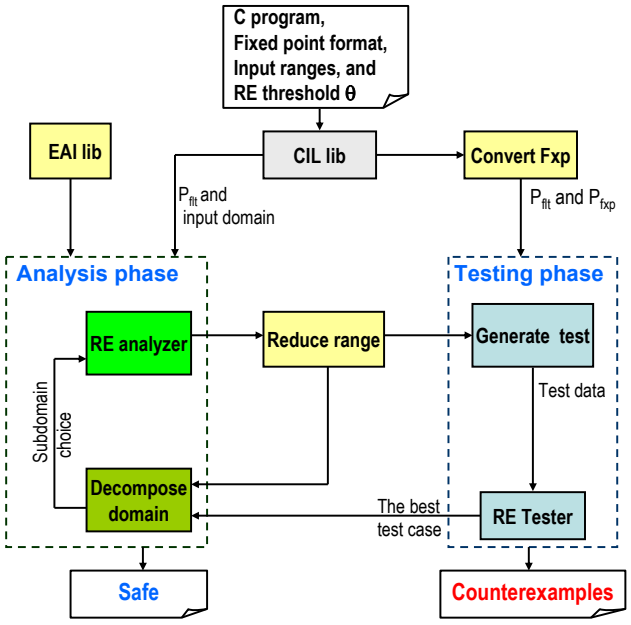


Fig. 11 CANAT system

Fig. 11 shows the construction of CANAT. CANAT uses three external tools:

- CIL library [5] as a preprocessor,
- Weighted PDS library [6] as a backend weighted model checking engine, and
- CANA [44] as an RE analyzer.

CANAT has 4 main modules as follows:

1. *Reduce range* module clarify: (1) the subdomains of input domain that contain maximum (or minimum) REs, (2) the choice of ticks for testing generation.
2. *Decompose domain* module divides the reduced domain (e.g., D_{max},D_{min}) into two subdomains.
3. *Generate Test* module generates set of test data T based on information obtained from *Reduce range* module.
4. *RE Tester* module automatically generates two programs corresponding to input C program, one uses fixed-point arithmetic, while the other uses floating-point arithmetic. Then, these two programs are executed with the set of test data T. The testing REs are the differences between the results of these two programs.

[5] http://hal.cs.berkeley.edu/cil/
[6] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/

Experimental results

Table 4 shows the results of checking 4 programs (on PC with Intel(R) Xeon(TM)CPU 3.60GHz, 3.37Gb of memory). *The first column* shows the names of 4 programs.

1. **P2** from Fig. 1: We set the initial range $[-1,3]\times[-10,10]$, $fp\in\{7,8,9,10\}$, and $\theta\in\{0.001+0.002i \mid 0\leqslant i\leqslant 9\}$.
2. **P5** that computes $P5(x)=1-x+3x^2-2x^3+x^4-5x^5$: We set the initial range $[0,1]$, $fp\in\{7,8,9,10\}$, and $\theta\in\{0.001+0.01i \mid 0\leqslant i\leqslant 9\}$.
3. **Sine** that computes the sine function using Taylor expansion up to degree 21: We set the initial range $[0,1]$, $fp\in\{7,8,9,10\}$, and $\theta\in\{0.001+0.005i \mid 0\leqslant i\leqslant 9\}$.
4. **pMpeg** that consists of one 8×8 loop taken from the mpeg4 decoder reference algorithm: We set the initial range $[0,30]$, $fp\in\{7,8,9,10\}$, and $\theta\in\{0.001+0.05i \mid 0\leqslant i\leqslant 9\}$.

We compare experimental results by

1. ORE analysis by CANA (CANA) followed by Randomly generated testing (**Random test**), and
2. Counter example guided narrowing loop by repeating ORE analyes (**CANAT anmalysis**) and refined testing (**CANAT test**).

Both try 40 settings (i.e., different numbers of digits in fixed-point numbers, different RE thresholds) for each program. For fair comparison, we make the total numbers of test cases to be the same for each setting; **Random test** generates 200 test cases, and (**CANAT test**) generates 20 test cases for each refinement loop, which is repeated 10 times.

%Checked columns show that how many percentage (among 40 settings) is detected either to be safe or ORE violation. Although the experiment remains a toy, it shows clear improvement of testing and analysis.

6 Related Work

The ORE problems are one of the central issues in the numerical analysis [17,13]. There are lots of works on mathemarical reasoning to estimate OREs [17,16], and there is a well-known methodology for the precise addition of floating numbers, which cancells the effect of REs [27]. It is extended to the precise multiplication [46], and recently verified numerical computation is evolving.

Our focus is more on static detections of OREs of programs, and we omit huge references of these areas, which are beyond scope of the paper.

Input program	CANA	Random test	Time (s)	%Checked	CANAT analysis	CANAT test	CANAT time(s)	%Checked of CANAT
P2	15	11	7	65.00%	20	18	13	95.00%
P5	9	15	14	60.00%	12	19	24	77.50%
Sine	19	7	37	65.00%	21	8	81	72.50%
pMpeg	11	11	65	55.00%	11	19	121	75.00%

Table 4 Experimental results of CANAT

Range representations of real numbers

Due to REs, we need to evaluate values of real numbers by some representation of ranges. They are classically classified into:

- Interval, which is the Cartesian product of one dimensional intervals [2, 40].
- Octagon, which is surrounded by either vertical, horizontal, or diagonal lines [39].
- Polyhedra, which is represented as the conjunction of linear inequalities [8]. Recently, its refinement SubPolyhedra was proposed [29] by reducing deduction rules among linear inequalities, yet preserving expressiveness.

We are more focus on intervals, and we call a range described by a pair of the lowest and the highest value by a classical interval (CI). CI is generalized to allow swapping of boundaries [18].

By introducing noise symbols, which preserve dependency of uncertainty, Affine interval (AI) has been proposed [54, 55]. Later, we will see how AI is applied as overapproximation for ORE analyses.

Extended Affine interval (EAI) has proposed for under approximation, based on the mean value theorem and Kaucher arithmetic [21]. EAI replaces real coefficients of AI with CI coefficients. We apply this idea for overapproximation of ORE analysis.

ORE analysis

For a static analysis, we need a concrete semantics. We obey the semantics of propagation of OREs to [32].

There are three kinds of OREs, caused by:

- real numbers to floating-point numbers conversion,
- real numbers to fixed-point numbers conversion, and
- floating-point numbers to fixed-point numbers conversion.

ORE analysis are mainly investigated for the first and the third.

For the real numbers to floating-point numbers conversion, ORE analysis adapt AI [20, 33] (which introduced widening operators to handle loops), except that the octagon abstract domain is used in [38].

They are implemented and showed experimental results. FLUCTUAT is presented in [20] and [38] showed a case study on an embedded avionics software. The technique of [33] is further applied on TMS320 C3X assembler programs [34].

These ORE analyses are overapproximation, and easily cause false positives. An ORE analysis with under approximation is proposed based on the mean value theorem and Kaucher arithmetic [21], to sandwich OREs from both sides. However, strictly speaking, this under approximation is for real number variables rather than floating-point number variables.

The floating-point numbers to fixed-point numbers conversion typically appears in hard-wired algorithms and/or embedded systems. Apart from difficulties in hardware encoding difficulties [3, 5, 25, 26, 37, 51, 52], there are strong demand to solve ORE problems.

For the floating-point numbers to fixed-point numbers conversion, Fang, et.al. proposed an ORE analysis based on AI, intended for DSP applications [9–11]. We are facing on the same problem, but with different intervals, EAI. In our implementation, we adapted a sophisticated weighted model checking, whereas they adapt direct bit-vector encoding. For scalability, they also applied probabilistic reduction of the search space.

Thanks to the problem nature, we can examine OREs by testing, since we can compute both floating-point numbers and fixed-point numbers. [56] showed a such testing tool.

We further combined an ORE analysis and testing by a counter example guided narrowing approach, which refines the focus of testing and avoid false positives in an early stage.

Numerical Constraint Solvers

Recently, several tools have been developed as variations of SMT to solve non-linear numerical constrains. For instance,

- iSAT [12], which evaluates non-linear operations to interval constraints by overfapproximation.
- minismt [57], which covers specific irrationals, such as rational numbers and roots of small integers. They are symbolically represented and its bounded search is encoded as CNF.
- a tool for Simulink/Stateflow models [24], which applied a variation of polyhedra, called the *bounded vertex representation* for under-approximation.

Ganai and Ivancic [14] introduced a new method to face with decision problems involving non-linear constraints on bounded integers. Each nonlinear operation is encoded into a Boolean combination of linear arithmetic constraints based on CORDIC algorithm. Then, the linearized formula will be input of a DPLL-style Interval Search Engine that explores various combination of interval bounds using a SMT solver.

To solve an interval constraint, they divide the input ranges to smaller ranges, which is similar to ours. Adding to the difference of target domains (i.e., bounded integers and floating/fixed-point numbers), the differences are, (1) we use EAI instead of CI, (2) combination with testing, and (3) weighted model checking instead of (SMT) solver.

To solve dataflow equations over infinite domains, such as numerical constraints (mostly on integers), several algorithms are proposed in the context of weighted pushdown model checking [19,15,30,41].

The library Apron of numerical abstract domains is also freely available [23].

Refining analyses and testing

As general setting of static detection, recently the refinement loop of analyses and testing is extensively investigated.

Counterexample Guided Abstraction-refinement (CEGAR) [6] is widely applied methodology, in which the initial abstract model typically nondeterministic control structures for conditional branches. When (possible spurious) counter examples are found, symbolic techniques refine the model by hooking more deterministic behavior.

Proofs from Tests [4] presented an algorithm DASH to check if a program satisfies a safety property. It uses only test generation, and it refines and maintains a sound program abstraction as a consequence of failed test generation operations. This enables us a light-weight refinement loop with neither any extra theorem prover calls nor any global may-alias information.

Our methodology of counter example guided narrowing tries to refine the focus of testing based on ORE analysis results. Fortunately, by the nature of EAI, ORE analysis results tell us which input parameter is dominant for REs. By using this information, we can effectively focus on the most problematic point.

7 Conclusion

Motivated by automatically detecting OREs of the hardwired conversion of DSP encoders/decoders, this paper proposed techniques for automatic detection of overflow and roundoff errors, caused by the floating-point number to fixed-point number conversion. Our contribution is summarized as follows.

- An extended affine interval (EAI) is newly proposed to overapproximate overflow and roundoff errors. EAI has two main advantages over current methods: EAI is more precise than CI and EAI forms are more compact than AI forms.
- An ORE analysis method based on weighted model checking is proposed and implemented as CANA.
- The counterexample-guided narrowing, in which an analysis and testing refine each other, is applied to the RE problem and implemented as CANAT. Thus, the result will be more precise than either testing or static analysis alone.

We also performed preliminary experiments of CANA and CANAT, which shows optimistic results.

For future works, one obstacle is the scalability of CANA and CANAT. We are optimistic since DSP algorithms (e.g., digital video compression [47]) are often compositional. They typically consist of sequences of computations with fixed-length arrays and bounded loops. Therefore, we can divide the algorithm to small fragments and check each separately.

Second, widening operator design. Currently, we did not introduce widening operators, but first focus on precision of ORE detection. (For instance, the widening operator [20] leads to inevitably lose precision a lot.) Its drawback is that the class of target programs has strong limitation, though it seems enough for the core part of DSP encoders/decoders.

We are also interested in an automatic correction of a program to improve with less OREs. For instance, in [36], there are several techniques to automatically improve numerical precision, such as:

- swapping the order of arithmetic operations,
- explicit shifting of the order of magnitude, and
- symbolic executions.

Our ORE analyzer, CANA, can generate the information about range values of fixed-point numbers and their RE at each point of the program. There information is helpful information to automatic source code correction.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Alefeld, G. and Herzberger, J.: Introduction to Interval Computations. Academic Press, N.Y. (1983)

3. Banerjee, P., Bagchi, D., Haldar, M., Nayak, A., Kim, V., and Uribe, R.: Automatic Conversion of floating-point MATLAB Programs into fixed-point FPGA Based Hardware Design. In *Proc. of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 263, IEEE Computer Society (2003)
4. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S.D., Thakur, A.V.: Proofs from Tests. In *IEEE Transactions on Software Engineering: Special Issue on the ISSTA 2008 Best Papers*, IEEE Computer Society (2010)
5. Belanovic, P. and Rupp, M.: Automated Floating-Point to Fixed-Point Conversion with the Fixify Environment. In *Proc. of the 16th IEEE International Workshop on Rapid System Prototyping*, pp. 172 - 178, IEEE Computer Society (2007)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H.: Counterexample-guided abstraction refinement. In *Proc. of the 12th International Conference on Computer Aided Verification* , pp. 154-169, Springer-Verlag (2000)
7. Cousot, P. and Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 238-252, ACM (1977)
8. Cousot, P. and Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.84-96, ACM (1978)
9. Fang, C.F., Rutenbar, R.A., and Chen, T.: Fast, accurate static analyis for fixed-point finite precision effects in DSP designs. In *Proc. of the International Conference on Computer Aided Design*, pp. 275-282, IEEE Computer Society (2003)
10. Fang, C.F.: Probabilistic Interval-Valued Computation: Representing and Reasoning about Uncertainty in DSP and VLSI Design. Ph. D. Diserration, Depterment of Electrical and Computer Engineering, Carnegie Mellon University (2005)
11. Fang, C.F., Rutenbar, R.A., Püschel, M. and Chen, T.: Toward efficient static analysis of finite-precision effects in DSP applications via Affine arithmetic modeling. In *Proc. of Design Automation Conference*, pp. 496-501, ACM Press (2003)
12. Franzle, M., Herde, C., Teige, T., Ratschan, S., and Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. In *Journal on Satisfiability, Boolean Modeling and Computation*, Vol 1, pp. 209-236 (2007)
13. Friedman, M. and Kandel, A.: Fundementals of Computer numerical analysis. CRG Press (1994)
14. Ganai, M.K and Ivancic, F.: Efficient Decision Procedure for Bounded Integer Non-linear Operations using SMT(LIA), In *Proc. of Haifa Verification Conference 2008*, LNCS, vol.5394, pp.68-83, Springer (2009)
15. Gawlitza, T. and Seidl, H.: Precise Fixpoint Computation Through Strategy Iteration. In *Proc. of the European Symposium on Programming 2007*,LNCS, vol. 4421, pp. 300315, Springer (2007)
16. Giraud, L., Langou, J., Rozloznik, M., and Eshof, J.V.D.: Rounding error analysis of the classical Gram-Schmidt orthogonalization process. *Numerische Mathematik*, v.101 n.1, pp.87-100 (2005)
17. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* , pp.5-48, ACM (1991)
18. Goldsztejn, A., Daney, D., Rueher, M., and Taillibert, P.: Modal intervals revisited: a mean-value extension to generalized intervals, In *Proc. of the 1st International Workshop on Quantification in Constraint Programming*, 2005.
19. Gopan, D.: Numeric program analysis techniques with applications to array analysis and library summarization. Ph. D Thesis, University of Wisconsin-Madison (2007)
20. Goubault, E. and Putot, S.: Static analysis of numerical algorithms. In *Proc. of the 13th International Static Analysis Symposium*, pp. 18-34, Springer-Verlag (2006)
21. Goubault, E. and Putot, S.: Under-approximations of computations in real numbers based on generalized affine arithmetic. In *Proc. of the 14th International Static Analysis Symposium*, pp. 137-152, Springer-Verlag (2007)
22. 754-2008 IEEE Standard for Floating-Point Arithmetic at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4610935. Accessed February 2010.
23. Jeannet, B., and Miné, A.: Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21st International Conference on Computer Aided Verification*, LNCS, vol.5643, pp. 661-667, Springer-Verlag (2009)
24. Kanade, A., Alur, R., Ivanc, F., Ramesh, S., Sankaranarayanan, S., and Shashidhar, K.C.: Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models. In *Proc. of the 21st International Conference on Computer Aided Verification*, LNCS, vol.5643, pp. 430-445, Springer-Verlag (2009)
25. Keding, H., Hurtgen, F., Willems, M., and Coors, M.: Transformation of Floating-Point into fixed-point Algorithms by Interpolation Applying a Statistical Approach. In *Proc. of the 9th International Conference on Signal Processing Applications and Technology*, pp. 270-276, ACM (1998)
26. Kim, S., Kum, K., and Sung, W.: Fixed-point optimization utility for C and C++ based digital signal processing programs. In *IEEE Transactions on Circuits and Systems II*, vol. 45, no.11, pp. 1455-1464, IEEE Computer Society (1998)
27. Knuth, D.E.: The Art of Computer Programming, Vol. 2: Seminumerical Algorithms. AddisonWesley (2002)
28. Lacey, D., Jones, N.D., Wyk, E.V., and Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. POPL, pp.283-294, ACM (2002)
29. Laviron, V. and Logozzo, F.: SubPolyhedra: A (more) scalable approach to infer linear inequalities. *Proc. of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, LNCS, vol. 5403, pp. 229-244, Springer-Verlag (2009)
30. Leroux, J. and Sutre, G.: Accelerated data-flow analysis. In *Proc. of the 14th International Static Analysis Symposium*, LNCS, vol. 4634, pp. 184199, Springer (2007)
31. Loh, E. and Walster, G.W.: Rump's example revisited. Realiable Computing, vol.8 (3), pp.245-248 (2002)
32. Martel, M.: Semantics of roundoff error propagation in finite precision calculations. In *Higher-Order and Symbolic Computation*, v19(1), pp.7-30, Springer Netherlands (2006)
33. Martel, M.: Static analysis of the numerical stability of loops. In *Proc. of the 9th International Static Analysis Symposium*, LNCS, vol. 2477, pp. 133-150, Springer-Verlag (2002)
34. Martel, M.: Validation of assembler programs for DSPs: A static analyzer. In *Proc. of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 8-13, ACM Press (2004)
35. Martel, M.: Semantics-Based Transformation of Arithmetic Expressions. In *Proc. of the 14th International Static Analysis Symposium*, LNCS, vol.4634 pp.298-314, Springer-Verlag (2007)
36. Martel, M.: Program transformation for numerical precision. In *Proc. of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation 2009* pp. 101-109, ACM Press (2009)
37. Menard, D., Chillet, D., Charot, F., and Sentieys, O.: Automatic floating-point to fixed-point conversion for DSP code

generation. In *Proc. of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 270-276, ACM (2002)
38. Mine, A.: Relational abstract domains for the detection of floating-point run-time errors. In *Proc. of the European Symposium on Programming (ESOP 2004)*, Vol. 2986, pp. 317, Springer (2004)
39. Mine, A.: The octagon abstract domain. Higher-Order and Symbolic Computation, Vol.19 (1), pp.31-100, 2006.
40. Moore, R.E.: Interval Analysis. Prentice-Hall (1966)
41. Mller-Olm, M. and Seidl, H.: Precise interprocedural analysis through linear algebra. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 330-341, ACM (2004)
42. Necula, G.C., McPeak, S., Rahul, S.P., and Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proc. of the 11th International Conference on Compiler Construction*, pp. 213-228, Springer-Verlag (2002)
43. Nielson, F., Nielson, H. R., Hankin, C.: *Principles of program analysis*. Springer (1998)
44. Ngoc, D.T.B. and Ogawa, M.: Roundoff and Overflow Error Analysis via Model Checking. In *Proc. of the 7th International Conference on Software Engineering and Formal Methods*, pp.105-114, IEEE Computer Society (2009)
45. Ngoc, D.T.B. and Ogawa, M.: Checking Roundoff Errors based on Counterexample-Guided Narrowing. In *Proc. of the 25th IEEE/ACM International Conference on Automated Software Engineering*, to appear as a short paper (2010)
46. Ogita, T., Rump, S.M., and Oishi, S.: Accurate sum and dot product. SIAM Sci. Comp, vol.26 (6), pp.1955-1988 (2005)
47. Peter, S.: Digital Video Compression. mcgraw-Hill (2004)
48. Reps, T., Schwoon, S., Jha, S., and Melski D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proc. of the 10th International Static Analysis Symposium*, pp. 189-213, Springer-Verlag (2003)
49. Sagiv, M., Reps, T., and Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. In *Theoretical Computer Science*, v.167 n.1-2, p.131-170 (1996)
50. Schmidt, D.A. and Steffen, B.: Program analysis as model checking of abstract interpretations. In *Proc. of the 5th International Conference on Static Analysis Symposium*, pp. 351-380, Springe-Verlag (1998)
51. Shi, C. and Brodersen, R.W.: An automated floating-point to fixed-point conversion methodology. In *Proc. IEEE International Conference on Acoust., Speech, and Signal Processing*, Vol. 2, pp. 529-532 (2003)
52. Shi, C.: Floating-point to Fixed-point Conversion. Ph. D. Thesis, University of California - Berkeley (2004)
53. Steffen, B.: Data Flow Analysis as Model Checking. In *Proc. of the International Conference on Theoretical Aspects of Computer Software*, p.346-365, Springer-Verlag (1991)
54. Stolfi, J.: Self-Validated Numerical Methods and Applications. Ph. D. Dissertation, Computer Science Department, Stanford University (1997)
55. Stolfi, J. and Figueiredo, L.H. de: An introduction to affine arithmetic. In *Tendencias em Matematica Aplicada e Computacional*, Vol.3(4), pp. 297-312 (2003)
56. Wijaya, S. and Cantoni, A.: A Java Simulation Tool for Fixed-Point System Design. In *Proc. of the 2nd International Conference on Simulation Tools and Techniques*, pp. 1-10, ACM (2009)
57. Zankl, H. and Middeldorp, A.: Satisfiability of non-linear (ir)rational arithmetic. *In Proc. of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, to appear (2010).