Combining Static Analysis and Testing for Overflow and Roundoff Error Detection

by

DO THI BICH NGOC

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Mizuhito Ogawa

School of Information Science
Japan Advanced Institute of Science and Technology

July 13, 2010

Abstract

In computer algorithms, real numbers are often represented as floating point numbers. However, hardware typically uses *fixed point number* representations for lower cost and higher speed. Direct transformations from a reference algorithm to a hardware algorithm with fixed point numbers often cause different computational results because of *overflow errors* and *roundoff errors*. Hence, when implementing hardware algorithms we need to consider: (1) whether overflow errors occur or not and (2) whether the roundoff errors exceed a predefined threshold bound or not.

The problem of detecting overflow and roundoff errors has been studied since 1960s, and is still an active research area. Originally, overflow and roundoff errors are often detected manually by using mathematical reasoning or testing. Recently, there are some extensive works of automatically checking overflow and roundoff errors based on static analysis and abstraction. In order to abstract overflow and roundoff errors, there are two well known techniques. The first technique is *Classical interval* (CI) which keeps the possible lowest and highest values as a segment. The second technique is *Affine interval* (AI) which introduces symbolic manipulations on noise symbols, to handle correlations between variables. AI arithmetic supplies higher precision than CI one. However, for nonlinear operations, AI arithmetic requires to introduce a fresh noise symbol each time. A question naturally raised is that: can we construct new interval arithmetic such that it is simpler than AI but as precise as AI?

It is worth emphasizing that static analysis is useful in automatically proving safety properties of programs but it may return spurious counterexamples due to approximation. In contrast, testing can return exact roundoff errors, while it virtually cannot cover all possible inputs. The challenge is: how to bridge the gap between testing and static analysis?

Motivated by the above questions, this thesis is concerned about the problem of detecting overflow and roundoff errors when converting floating point numbers to fixed point numbers for a class of C programs with bounded loops, fixed size arrays, and no pointer manipulations. This class of programs is sufficient to capture the core algorithms of DSP encoders/decoders. The contributions and achievements of this thesis are summarized as follows:

- New intervals: improve current intervals (i.e., CI and AI) to approximate overflow and roundoff errors. Two new intervals named "Extended affine interval" (EAI) and "Positive-noise affine interval" (PAI) are proposed. EAI represent ranges as AI form

whose each noise symbol is assigned a CI. By this means, the results of nonlinear operations can be approximated to keep linear form without introducing new noise symbols as AI does. In positive-noise AI, the noise symbols in PAI lie in [0,1] (instead of [-1,1]) and the nonlinear operations are designed based on Chebyshev approximation to improve the precisions.

- Overflow and roundoff error analysis as weighted model checking: propose and implement the overflow and roundoff error analysis based on weighted model checking. The overflow and roundoff error abstractions based on intervals (CI, AI, and EAI) are used to create sets of weights. Then, a C program is modeled by a weighted transition system (a finite transition system + weight domain), where weight domain is generated by an on-the-fly manner. Finally, the overflow and roundoff error problem was reduced to checking reachability properties for the weighted transition system. The proposed framework is implemented in an automatic overflow and roundoff error analyzer, called CANA (C ANAlyzer). Experimental results on small programs show that the EAI is much more precise than CI.

- Combine static analysis and testing for roundoff error detection: propose and implement a hybrid approach called "counterexample-guided narrowing" in which analysis and testing refine each other. This approach is applied to improve the precision of roundoff error analysis and implement the proposed framework as a prototype tool CANAT (C ANAlyzer and Tester). Although our experiments are still small, the results outperforms both random test and static analysis.

Key words: software verification, static analysis, model checking, testing, roundoff error, overflow error, affine interval.

Acknowledgments

This thesis would not have been possible without the help of many people. First of all, I would like to express my deep gratitude to my principal supervisor, Professor Mizuhito Ogawa. He has been a good and patient advisor. I learned much from him how to be a researcher. It would be impossible to pinpoint his contributions large and small to this work; his encouragement has been very important as well. Without him, this thesis would simply not exist.

That the thesis does not exist in the present form is due also to the enthusiastic interest, support, and criticism I have received from my colleagues and members of my reading thesis committees, Professor. Kokichi Futatsugi, Professor. Shin Nakajima, Professor. Kazuhiro Ogata, Professor. Toshiaki Aoki, and Dr. Hirokazu Anai. I have greatly benefited from their guidance and helpful comments. This thesis was markedly improved because of their critical reading and valuable suggestions. Especially, I would also like to give special thanks to Professor Toshiaki Aoki for his supervision of my sub-theme research.

The faculty, staff and students at the Software Verification Laboratory have provided an excellent academic environment; in particular, Associate Professor Fumihiko Asano, Assistant Professor Nao Hirokawa, Dr. Li Xin, Dr Li Gouqiang, Dr Nguyen Van Tang, Mr Klein Dominik, Mr Song Lin, Mr To Van Khanh. Our technical discussions have helped in my work, while our more philosophical discussions have been thoroughly enjoyable. Especially, I would like to thank Nao for his valuable guidance and technical supports to this thesis.

My Vietnamese friends have made my stay in JAIST fun, exciting and memorable. I would like to thank my best friends Nguyen Quang Huy, Pham Gia Vinh Anh who always encourage and support me in research whenever I need. I thank the teachers in the Technical Communication Program Office - who helped me to refine my technical drafts of papers and thesis.

Finally, I would also like to take this opportunity to thank my family for their love and supports; especially, for my parents for encouraging me, and for my husband for being my best friend and supporting me. I dedicate this thesis to my son Nguyen Nhat Hung, who provided me with many joyous moments. Whenever I needed to clean my head from all this verification stuff he was ready to give me plenty of other things to do. Without the joy of having him I am not sure I would have completed this task.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Sources of Numerical Errors

In general, a numerical algorithm for solving a given problem may have errors of one or several types. Although different source initiate the error, they all cause the same effect: diversion from the exact answer. Some errors are small and may be neglected. Others may be devastating if overlooked. The major sources of errors are as follows.

1. **Human error**: is introduced during the process of solving the problem by human, for example, changing signs in a formula or a simple programming error.

2. **Truncation error**: occurs when we are unable to evaluate explicitly a given quantity, and replace it by an approximation that can be computed.

   For example, the function $sin(x)$ can be replaced by

   $$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + ...$$

   We may calculate $(x - \frac{x^3}{3!} + \frac{x^5}{5!})$ as an approximation to $sin(x)$, provide the error term $E(x) = -\frac{x^7}{7!} + ...$

3. **Machine error**: A floating point (or fixed point) number processed by a computer may not have an exact representation. Also, floating point (or fixed point) arithmetic in general is not exact. For example, if the length of a mantissa is 4 bits, then $0.1101 \times 0.1011 = 0.1000$ (chopping mode) whereas the exact values of the left-hand side is 0.10001111. There are two types of machine errors : *roundoff error* and *overflow error*.

4. **Inaccurate observation**: Many numerical processes involve physical quantities such as the speed of light, density of iron, or the constants of gravity. These quantities are provided by experiments and naturally introduce some experimental errors.

For example, the speed of light in vacuum is

$$c = (2.997925 + \epsilon) \times 10^{10} cm/sec. |\epsilon| \leqslant 10^{-6}$$

Experimental or observational errors cannot be removed or even reduced, without improving the observational technique.

5. **Modeling error**: Constructing a mathematical model is the first step in the process of solving a problem. However, an equation or a system of equations that is expected to describe some phenomena will generally only approximate the physical reality. Occasionally a mathematical model may be solved successfully, and yet the computed and the experimental results are far apart. This may occur if an important physical aspect is overlooked and the mathematical model is unjustly simplified.

Truncation errors, inaccurate observations, and modeling errors are caused by human when create numerical algorithm from real problem. Thus, they are often analyzed manually. Human error and machine error are caused by implementing the algorithm in computer (or hardware system). While human error can be avoided by carefully code the algorithm, machine error cannot be avoided because of finite presentation of real numbers. In this thesis, we focus on automatically analyzing machine error (i.e., overflow and roundoff errors) when converting floating point numbers to fixed point numbers.

1.2 Overflow and Roundoff Errors Problem

In the computers, the (infinite) real numbers are approximated by finite numbers (e.g., floating point numbers, fixed point numbers). Because of finite representation, the overflow and roundoff errors (OREs) may occur. There are three kinds of OREs:

1. Real numbers vs floating point numbers: approximating real numbers as floating point numbers causes OREs. These OREs often appear in computers since most of them use floating point arithmetic. Most of ORE researches focus on this kind of OREs [13, 26].
2. Real numbers vs fixed point numbers: OREs may occur when real numbers are approximated as fixed point numbers. This kind of errors appears when implement a (reference) algorithm with real numbers in a hardware with fixed point numbers [32].
3. Floating point numbers vs fixed point numbers: many algorithms implemented in computers are proved to be satisfied the ORE requirement. We also want to re-implement them in hardware systems or devices, such as, PDA, mp3 players, videogame consoles. These devices need to convert the floating point numbers to

fixed point numbers for lower cost and higher speed. Recently, there are many works focus on converting floating point numbers to fixed point numbers [3, 5, 44, 58, 59]. Because the floating point numbers can represent more precise values than that of fixed point numbers, the conversation from floating point numbers to fixed point numbers may causes OREs.

The OREs will be propagated through computations of the program. Further, the computations themselves also 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). OREs sometimes are propagated too much and may cause serious problems. One example about disaster causes by roundoff error is "The Patriot Missile Failure" [1].

Example 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.*

The reason is roundoff error when representing real number 1/10 *by fixed point number. The number* 1/10 *equals* $1/24+1/25+1/28+1/29+1/212+1/213+....$ *In other words, the binary expansion of* 1/10 *is* 0.0001100110011001100110011001100.... *Now the* 24 *bit register in the Patriot stored instead* 0.00011001100110011001100 *introducing an error of* 0.0000000000000000000000011001100... *binary, or about* 0.000000095 *decimal. Multiplying by the number of tenths of a second in 100 hours gives* $0.000000095\times100\times60\times60\times\ 10 = 0.34$. *A Scud travels at about 1,676 meters per second, and thus travels more than half of kilometer in this time.*

Another example about disaster cause by overflow error is "The Explosion of the Ariane 5" [2].

Example 2 *On 1996 June 4, an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The destroyed rocket and its cargo were valued at* $500 *million.*

The cause of the failure was a software error in the inertial reference system. Specifically a 64 *bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a* 16 *bit signed integer. The number was larger than* 32767*, the largest integer storable in a* 16 *bit signed integer, and thus the conversion failed.*

Target programs

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

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 [54]. 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. 1.1)

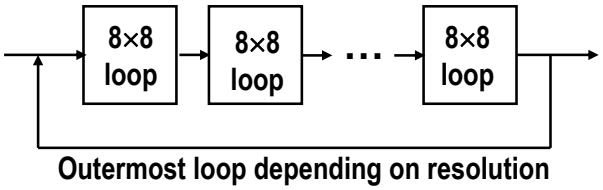


Figure 1.1: Typical loops in Mpeg decoder

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,

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 **"satisfies the ORE requirement"** if for all inputs, there is no OEs and REs of the result lie in $[-\theta, \theta]$.

Example 3 *Fig. 1.2 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* $[-0.26,\ 0.26]$*?*

```
    /* 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;    }
```

Figure 1.2: An example of a C program

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).

Example 4 *The function maintest in Figure 1.2 takes two parameters x, y and returns rst . Assume that the initial ranges of x, y are $[-100, 100], [-100, 100]$, respectively.*

We need to convert the floating point type to fixed point type in that the width of integer part is 11 and the width of fraction part is 4. The conversion is called "satisfying ORE requirement" if there are no overflow error and the roundoff error of rst that lies in $[-0.1, 0.1]$.

Floating point function *Let us consider one input: $(x = 99.03, y = 100)$. Because $x > 0$, the (floating point) function maintest will return $rst = x * x - y = 9706.9409$.*

Fixed point function *The corresponding fixed point input is $(x_{fx} = 99.00, y_{fx} = 100)$. Hence, the roundoff error of x_{fx} is $r_x = 0.03$ and the roundoff error of y_{fx} is $r_y = 0$. The (fixed point) function maintest will return $rst_{fx} = x_{fx} * x_{fx} - y_{fx} = 9701$. It means that the roundoff error of rst is $r_{rst} = rst - rst_{fx} = 5.9409$. This value lies out $[-0.1, 0.1]$. Furthermore, at statement (3), we have $rst_{fx} = x_{fx} * x_{fx} = 9801 > 2^{11}$, hence, the overflow error occurs at this location. As a result, this conversion does not "satisfy the ORE requirement".*

1.3 The Existing Approaches

OREs are sources of serious bugs that affect economy, or even cause deaths. Therefore, ORE detection has been attracted many attention over the past fifty years (since 1960s) [21, 26, 13, 14, 45, 51]. There are main existing approaches for these problems such as mathematical reasoning, numerical analysis, and testing. In the following, we are going to give a survey on these main approaches.

Mathematical reasoning

In the mathematical reasoning [21], the user approximates an ORE formula by inequations and mathematical transformations. Mathematical reasoning method normally returns precise results but user may take monthly to solve the ORE formula.

Testing OREs

The OREs when approximating real numbers to floating point (or fixed point) numbers cannot be tested automatically. The reason is the computer cannot represent (infinite) real numbers. However, the OREs between program with floating point arithmetic (P_{flt}) and the corresponding program with fixed point (P_{fxp}) can be computed [63]. An overflow error will occur when the result of a fixed point operation in P_{fxp} exceeds the range that fixed point number can represent. A (*true*) roundoff error is the difference between the execution of P_{flt} and P_{fxp}. The ORE testing problem is: whether there is an input such that OREs occur or its roundoff error does not lie in a given roundoff error threshold bound $[-\theta, \theta]$? If there exists such an input, we call it a *counterexample*.

Test data depends on both the input domain and the strategy of generating test data. For example, assuming that a program has 5 variables, each variable is initiated with range $[0, 255]$. For each variable, we choose 4 instances for both the fixed point part and roundoff error part. Then, the number of test cases is $4^{5\times 2}$ (about one million). This is a huge number, and testing becomes infeasible. The challenge of testing method is how to generate a set of test data that can cover all cases of the program with a small number of tests.

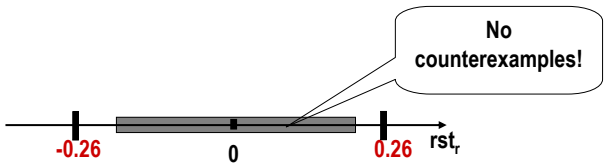
Static analysis OREs

From the viewpoint of abstract interpretation [9, 57], a static analysis determines ORE properties of a program by exhausting executions under abstraction, which propagates the ranges of both fixed point parts and roundoff errors from the begin to the end of the program [26, 51]. There are two main techniques to represent ranges. The first method uses *classical interval* (CI) [2, 47], which keeps the possible lowest and highest values as a segment. This method is simple but imprecise, because it does not handle

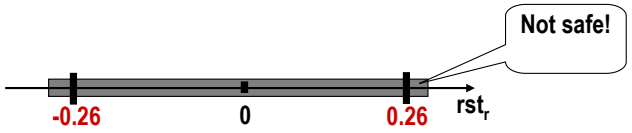
the correlations among variables. The second method uses *affine interval* (AI) [61, 62], which introduces symbolic manipulations on noise symbols, to handle correlations between variables. AI arithmetic supplies higher precision than CI one. However, for nonlinear operations (e.g., multiplication), AI arithmetic requires to introduce a fresh noise symbol each time. This leads to high complexity if there are many nonlinear operations. In order to improve efficiency and precision of the analysis, a question naturally raised is that: can we construct a new interval arithmetic such that it is better than CI and AI? Further, the over approximations occur at the control flows (such as the conditional branches and loops) and operations of the programs. Another question is how to reduce this over approximation.

Testing versus static analysis

We may face a situation that an roundoff error analysis reports that the roundoff error of the result exceeds the roundoff error threshold bound, but a test cannot find any counterexamples. The following example shows the problems of both testing and analysis:



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



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

Figure 1.3: Results of analyzing and testing C program in Figure 1.2

Example 5 *Assume that for the program in Fig. 1.2, 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* satisfies the ORE requirement *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. 1.3 a).*

The ORE analysis (in Chapter 6) reports that the roundoff error of `rst` *lies in* $[-0.28, 0.28]$*, which exceeds the roundoff error threshold* $[-0.26, 0.26]$ *(Fig. 1.3 b).*

Then, both testing and analysis cannot clarify whether the program in Fig. 1.2 satisfies the ORE requirement.

The challenge now is how to fill the gap between testing and static analysis. Remark that the smaller input ranges (of both the fixed point parts and the roundoff error parts) will produce a smaller ranges of the roundoff errors of results. Thus, we have an opportunity to exclude spurious counterexamples by refining input ranges and repeating executions of roundoff error analysis. A natural question is: can we combine testing and static analysis to exploit the advantages of these both methods?

- An roundoff error analysis result may show suspicious spots of input ranges, such as, which input variable affects the most, etc. This will give a focus of test data generation.
- A testing may show which spot of the input ranges is likely to maximize the roundoff error of the result. This will give a focus of both an roundoff error analysis and testing next round after input domain decomposition.

1.4 The Proposed Approach and Contributions of the Thesis

The aim of this thesis is to develop techniques to *automatically* detect OREs of a subclass C programs. The new techniques are intended to *safely* estimate OREs with *high precision*.

- First, we propose two new interval arithmetics, Extended affine interval (EAI) arithmetic and Positive-noise affine interval (PAI) arithmetic, which are useful in approximating OREs. EAI is extended from AI by assigning for each noise symbol a CI coefficient. Unlike AI, EAI nonlinear operations are defined without introducing new noise symbols. Thus, EAI has two main advantages over current methods. First, EAI can store information sources of uncertainty, whereas CI cannot. Second, EAI arithmetic does not introduce new noise symbols, while AI arithmetic does. PAI is another way to extend AI in that the noise symbols are set to lie in $[0, 1]$ (instead of $[-1, 1]$) and PAI nonlinear operations are defined to based on Chebyshev approximation to improve the precisions.

- Second, we propose an ORE analysis framework based on weighted model checking. In particular, the ORE abstraction based on ORE propagation rules and range representations (CI, AI, and EAI) are used to create the sets of weights. Next, the C program is modeled by a transition system in that the loops are unfolded as sequences of statements. Here, we only face with the programs with bounded-loops

(which are often appear in the hardware algorithms), hence the transition system will be finite. (Until now, we still not face with infinite programs because widening operation is required, thus, the analysis may be often over approximate too much, while the precision properties is very important in the OREs problem.) The weighted transition system is then generated as finite transition system + weight domain, where weight domain is generated in an on-the-fly manner. Finally, the ORE problem is reduced to checking reachability properties for the weighted transition system. We implement the proposed framework in an ORE analysis tool, called CANA (C ANAlyzer).

- Third, we propose a hybrid approach called counterexample-guided narrowing, which combines static analysis and testing for roundoff error detection. The result of analysis (in EAI form) provides useful information for testing phase (e.g., variables are irrelevant to roundoff errors of the results, variables affect the roundoff errors the most, and the ranges of inputs are most likely to cause the maximum roundoff error). These observations effectively narrow the focus of test data generation. In case testing does not find a witness of roundoff error violation, the analysis may be over approximate 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. We implement the proposed framework as an automatic prototype tool CANAT (C ANAlyzer and Tester), to detect roundoff errors of the programs which are converted from floating point numbers to fixed point numbers.

1.5 Structure of the Thesis

The remainder of the thesis is organized as follows:

- Chapter 2 introduces OREs problem and formalizes the ORE arithmetics.
- Chapter 3 introduces weighted model checking and how to treat dataflow analysis problem as weighted model checking and abstraction.
- Chapter 4 introduces two well known intervals, CI and AI. Sections 4.3 and 4.4 proposes two new intervals (i.e., EAI and PAI). Section 4.5 present how to implement above intervals on computers.
- Chapter 5 represents the ORE abstract domain based on ORE arithmetic and intervals.
- Chapter 6 proposes an ORE analysis approach as weighted model checking and ORE abstraction. An ORE analyzer, CANA, is also presented in this chapter.

- Chapter 7 proposes the counterexample-guided narrowing approach to detect REs and its implementation, CANAT.
- Chapter 7 discusses about related works.
- Chapter 8 closes the thesis with conclusions and discussions about the future works.

Chapter 2

Representation of Real Numbers in Computer and the ORE Problem

We first present overflow and roundoff error (ORE) problem when represent real numbers in computers such as floating point numbers (Section 2.1) and fixed point numbers (Section 2.2). Second, in Section 2.3 we introduce ORE arithmetics, which decomposes a number into a pair of floating point (or fixed point) and a roundoff error evaluation. Finally, we shows a method to compute the ORE constraints (a system of equations over program variables and a given threshold) of a program via using weakest preconditions in Section 2.4 and shows that solving ORE constraints is double exponential time class.

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. In general, we define a floating point number as follows [28]:

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

$$x = (-1)^s \times m \times b^e \tag{2.1}$$

*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 (i.e., $d_0 \neq 0$).*

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 2.1: The formats of floating point numbers

Example 6 *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 [28], shown in Table 2.1. 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 significand, and *overflow error* may occur due to the finite exponent.

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 [28].

- *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.

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 7 *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*

$$re_{fl}(x) = 0.0000000333333333333333333333333....$$

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 $(-1)^0 \times 0.00000065410^5 = 0.0654$.

Floating point multiplication

$e = 3$	$m =$	*4.734612*
$\times e = 5$	$m =$	*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 8 *In the IEEE 754 decimal32 format ($b = 10, p = 7, emax = 96$),*

$$\begin{array}{rl} & e = 48 \quad m = \quad 4.734612 \\ \times & e = 48 \quad m = \quad 5.417242 \\ \hline & e = 96 \quad m = 25.648538980104 \text{ (true product)} \\ & e = 96 \quad m = 25.64854 \text{ (after rounding)} \\ & e = 97 \quad m = \quad 2.564854 \text{ (after normalization)} \end{array}$$

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:*

$$a = spa_1a_2 \ldots a_{ip} \cdot a_{ip+1} \ldots a_{ip+fp},$$

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 9 *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

Fixed point format is the simple representation of real numbers. The OREs not only occur when converting real numbers to fixed point number, they also occurs when converting floating point number to fixed point numbers.

Roundoff error (RE) If the fraction part of number x 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

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

Example 10 *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

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

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 11 *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 (i.e., 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* $\boxed{*} = \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ *is defined below.*

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

Because of RE, the result of fixed point conditional expression is sometimes different from the result of real conditional expression. Therefore, the fixed point program leads to incorrect results. 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 value is $[x_f - |x_r|, x_f + |x_r|]$. 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.

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

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

Remark 2 *Other comparison operators (e.g.,* $>$, $!=$*) can be defined using the above operators.*

Example 12 *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, RE exceeds RE threshold, or we can conclude this computation does not satisfy the ORE requirement.

- ***Substraction**:*

$$\begin{aligned}(x_f, x_r)\boxminus(y_f, y_r) &= (rd_{fp}(34.56 - 98.76),\ 0.0078 - 0.00543 + re_{fp}(34.56 - 98.76)) \\ &= (-64.20,\ 0.00273)\end{aligned}$$

That means the result of substraction is -64.20 and its RE is $0.00273 < \theta$. We can conclude this computation satisfies ORE requirement.

- ***Multiplication**:*

$$\begin{aligned}(x_f, x_r)\boxtimes(y_f, y_r) &= (rd_{fp}(34.56 \times 98.76),\ 0.0078 \times 98.76 + 34.56 \times 0.00543 \\ &\qquad + 0.0078 \times 0.00543 + re_{fp}(34.56 \times 98.76)) \\ &= (3413.14,\ 0.963631154)\end{aligned}$$

That means the result of multiplication is 3413.14 ($> 10^3$) and its RE is $0.963631154 > \theta$. Thus, OE occurs and RE exceeds RE threshold. We then conclude this computation does not satisfy ORE requirement.

- ***Division**:*

$$\begin{aligned}(x_f, x_r)\boxdiv(y_f, y_r) &= (round_{fp}(34.56 \div 98.76),\ (34.56 + 0.0078) \div (98.76 + 0.00543) \\ &\qquad - 34.56 \div 98.76 + re_{fp}(34.56 \div 98.76)) \\ &= (0.34,\ 0.009998982437478376796415506923829 5)\end{aligned}$$

The result of division is 0.34 and its RE is $0.009998982437478376796415506923829 5$ ($< \theta$). We can conclude this computation satisfies ORE requirement.

2.3.2 Real-to-Float ORE Arithmetic

Similar to the above cases, we can define the rule to propagate OREs when converting real numbers to floating point numbers. 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* $\boxed{\circledast} = \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ *is defined below.*

$$
\begin{aligned}
(x_f, x_r)\boxplus(y_f, y_r) &= (rd_{fl}(x_f + y_f),\ x_r + y_r + re_{fl}(x_f + y_f)) \\
(x_f, x_r)\boxminus(y_f, y_r) &= (rd_{fl}(x_f - y_f),\ x_r - y_r + re_{fl}(x_f - y_f)) \\
(x_f, x_r)\boxtimes(y_f, y_r) &= (rd_{fl}(x_f \times y_f), x_r \times y_f + x_f \times y_r + x_r \times y_r + re_{fl}(x_f \times y_f)) \\
(x_f, x_r)\boxdiv(y_f, y_r) &= (rd_{fl}(x_f \div y_f), (x_f + x_r) \div (y_f + y_r) - x_f \div y_f + re_{fl}(x_f \div y_f))
\end{aligned}
$$

Real-to-Float ORE comparison operators when converting real numbers to floating point numbers are defined as follows:

Definition 8 (Real-to-Float 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.

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

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

Remark 3 *Other comparison operators (e.g.,* $>$, $!=$*) can be defined using the above operators.*

2.3.3 Float-to-Fixed ORE Arithmetic

The RE of the conversion from floating point numbers to fixed point numbers will be computed based on the REs when converting from real numbers to floating point numbers and converting from real numbers to fixed point numbers. 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 9 (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.*

$$
\begin{aligned}
(x_f, x_r)\boxplus(y_f, y_r) &= (rd_{fx}(x_f + y_f),\ x_r + y_r + re_{fx}(x_f + y_f) - re_{fl}(x + y)) \\
(x_f, x_r)\boxminus(y_f, y_r) &= (rd_{fx}(x_f - y_f),\ x_r - y_r + re_{fx}(x_f - y_f) - re_{fl}(x - y)) \\
(x_f, x_r)\boxtimes(y_f, y_r) &= (rd_{fx}(x_f \times y_f), x_r \times y_f + x_f \times y_r + x_r \times y_r + re_{fx}(x_f \times y_f) - re_{fl}(x \times y) \\
(x_f, x_r)\boxdiv(y_f, y_r) &= (rd_{fx}(x_f \div y_f), (x_f + x_r) \div (y_f + y_r) - x_f \div y_f + re_{fx}(x_f \div y_f) - re_{fl}(x \div y))
\end{aligned}
$$

Float-to-Fixed ORE comparison operators when converting floating point numbers to fixed point numbers are defined as follows:

Definition 10 (Float-to-Fixed ORE comparison operators) *Let* (x_f, x_r), *and* (y_f, y_r) *be representations of two floating point numbers* $x,\ y$.

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

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

Remark 4 *Other comparison operators (e.g.,* $>$, $!=$*) can be defined using the above operators.*

An instance of transformation

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 9.

Let us modify Definition 9 for transformation from floating point IEEE 754 binary64 $(2, 53, 1024)$ to fixed point $(2, ip, fp)$. Assume that the rounding mode is "round to nearest".

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 have:

$$\begin{cases} |\delta_+| & < & 2^{-38-fp} \\ |\delta_-| & < & 2^{-38-fp} \\ |\delta_\times| & < & 2^{fp-1} - 2^{2\times fp-1} + 2^{-38-fp} \\ |\delta_\div| & < & 2^{fp-1} + 2^{-38-fp} \end{cases} \tag{2.2}$$

Definition 11 (Float64-to-Fixed16 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. *Float64-to-Fixed16 ORE arithmetic* $\boxtimes = \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ *is defined below.*

$$\begin{aligned} (x_f, x_r) \boxplus (y_f, y_r) &= (rd_{fp}(x_f + y_f),\ x_r + y_r + \delta_+) \\ (x_f, x_r) \boxminus (y_f, y_r) &= (rd_{fp}(x_f - y_f),\ x_r - y_r + \delta_-) \\ (x_f, x_r) \boxtimes (y_f, y_r) &= (rd_{fp}(x_f \times y_f), x_r \times y_f + x_f \times y_r + x_r \times y_r + \delta_\times) \\ (x_f, x_r) \boxdiv (y_f, y_r) &= (rd_{fp}(x_f \div y_f), (x_f + x_r) \div (y_f + y_r) - x_f \div y_f + \delta_\div) \end{aligned}$$

where $\delta_+, \delta_-, \delta_\times, \delta_\div$ *are given in Equation 2.2.*

2.4 ORE Constraints of the Programs

To consider whether the RE of result of a concrete program lies within threshold bound or not, we firstly create the RE constraint based on the the initial range of input and the

threshold bound of RE. Next, we need to solve the RE constraint to clarify whether the RE occur or not.

A sample program language

A general program basically includes three types of instructions: assignments, conditions, and loops. This program language does not contain exception, recursive function and pointers.

We define the syntax of the core language as in Table 2.4.

Program	:=	Block
Block	:=	Stm;
		|Stm;Block
E	:=	c (constants)
		$\|x$ (variables)
		$\|E\ op\ E$ (operator $+, -, \times, \div$)
B	:=	$E\ cp\ E$ (compare operator $==, <, >, and, or, not$)
Stm	:=	$x = E$
		$\|If\ B\ then\ Block\ else\ Block$
		$\|\ while\ n\ do\ Block$

Table 2.2: Syntax of core language

In fact, we focus on analyzing the programs which are implemented in the hardware (e.g., Mpeg decoder algorithm), thus, this limitation of programming language are not affect to. Further, we does not analyze original C language, only it three address codes (CIL codes) which can be extended from this core languages.

Create ORE constrains based on Weakest precondition technique

Program verification based on weakest precondition The RE constraint can be generate directly by using weakest precondition technique [11]. Let us assume we want to verify a program where we know the postcondition but not the precondition:

$$\{?\}S\{R\}$$

In general, there could be arbitrarily many preconditions Q which are valid for the program S and a postcondition R. However, there is one precondition Q that describing the maximal set of possible initial inputs such that the execution of S leads to a state satisfying R. This Q is called the *weakest precondition*.

Formally, weakest-preconditions are defined recursively over the abstract syntax of statements. Given a program $s_1; s_2; \cdot; s_n$ with precondition$\{P\}$ and postcondition $\{R\}$, weakest precondition strategy to verify this program is started from s_n and $\{R\}$. We

produce $\{P_{n-1}\}$ is the weakest precondition for the statement s_n. $\{P_{n-1}\}$ now becomes the postcondition for s_{n-1}. Then we continue produce $\{P_{n-2}\}$ is weakest precondition of s_{n-1}.

$$\{P\}s_1; s+2; \cdot; s_n\{R\}$$

$$\{P\}$$

$$\{P_0\}$$

$$\{s_1\}$$

$$\{P_1\}$$

$$\cdot$$

$$\{s_{n-1}\}$$

$$\{R\}$$

Doing similarly, we obtain P_0 finally. What remains is to prove

$$P \Rightarrow P_0$$

Create ORE constraints The weakest preconditions for RE constraint is defined as Table 2.4.

$wp(x = E1 + E2, C)$	$=$	$C[(x_f, x_r)/(E1_f, E1_r) \boxplus (E2_f, E2_r)]$
$wp(x = E1 - E2, C)$	$=$	$C[(x_f, x_r)/(E1_f, E1_r) \boxminus (E2_f, E2_r)]$
$wp(x = E1 \times E2, C)$	$=$	$C[(x_f, x_r)/(E1_f, E1_r) \boxtimes (E2_f, E2_r)]$
$wp(x = E1 \div E2, C)$	$=$	$C[(x_f, x_r)/(E1_f, E1_r) \boxdiv (E2_f, E2_r)]$
$wp(if\ B\ then\ Block1\ else\ Block2, C)$	$=$	$(\widehat{B} \Rightarrow wp(Block1, C)) \wedge (\neg\widehat{B} \Rightarrow wp(Block2, C))$
$wp(stm; Block, C)$	$=$	$wp(Stm, wp(Block, C))$
$wp(while\ n\ do\ Block, C)$	$=$	$wp(Block \cdots (n\ times) Block, C)$

Table 2.3: Weakest precondition for ORE problem

Thus, for a given algorithm P, RE constraints for P is computed by:

1. the precondition (called R) are the constraints of forms $l_i \leqslant v_i \leqslant h_i$, where $[l_i, h_i]$ is the initial range of variable v_i.

2. the postcondition (called T) are the constraints of forms $|result_e| \leqslant \theta$, where θ is the threshold of RE of result.

3. the RE constraint for the algorithm P will be:

$$\forall v_1 ... v_n . R \Rightarrow wp(P, T) \tag{2.3}$$

If 2.3 is true then the RE value is acceptable. Otherwise, there is a input that makes the RE value too large. The problem now is how to verify the constraint (2.3).

Solving ORE constraint problem

We consider the case: the fixed point part is fixed, and the RE part is changeable. We have an observation that, even with this simple case, it will be reduced to polynomial inequations over real. Brown [6] shows that the quantifier elimination of polynomial inequations over real are double exponential. Therefore, the problem of solving requires doubly exponential time complexity. As a consequence, finding exactly ORE is impractical.

Chapter 3

Dataflow Analysis as Weighted Model Checking

It has been suggested intimate connections between dataflow analysis and model checking [48, 57]. 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, 35]. 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 [48], affine relation analysis [56], or ORE constraint analysis [51] 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 [55] 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). In this chapter, we briefly describe how to transform dataflow analysis problem as a model checking problem and abstraction (Section 3.1). We then describe the idea of transforming dataflow analysis problem to weighted model checking problem and abstraction in Section 3.2.

3.1 Dataflow Analysis as Model Checking and Abstraction

3.1.1 Dataflow Analysis

Dataflow analysis [50] is a special case of program analysis which statically computes information about the flow of data for each program point. This information must be a safe approximation of the desired properties of the run-time behavior of the program during each possible execution of that program for all possible inputs. In general, there are 3 common approach for solving dataflow analysis problem:

- *Constraint resolution systems:* consist of a constraint store and a logic for solving constraints. In particular, a program component constraints the semantic properties. These constraints are expressed in form of inequalities and the semantics properties are derived by finding a solution which satisfies all the constraints. In this approach, SAT/SMT solver can be used to help solving the constraints [19].

- *Model checking:* the classical dataflow analysis can be transformed to model checking problem by creating suitable abstractions of the programs as models and expressing desired properties in terms of boolean formulae. A model checking algorithm then discovers the states in the model that satisfy the given formulae.

- *Abstract interpretations* use abstraction functions to map the concrete semantic values to abstract semantics, perform the computations on the abstract semantics, and use concretization functions to map the abstract semantics back to the concrete semantic.

Dataflow analysis can be characterized by the following properties:

- Context sensitive: analysis is an interprocedural analysis that considers the calling context when analyzing the target of a function call. In particular, using context information one can jump back to the original call site, whereas without that information, the analysis information has to be propagated back to all possible call sites, potentially losing precision. In general, fully context sensitive analysis is very inefficient and most practical algorithms employ a limited amount of context sensitive.

- Flow sensitive: analysis takes into account the order of statements in a program. For example, a flow-insensitive pointer alias analysis may determine "variables x and y may refer to the same location", while a flow-sensitive analysis may determine "after statement 20, variables x and y may refer to the same location".

- Path sensitive: analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions. For instance, if a branch

contains a condition x¿0, then on the fall-through path, the analysis would assume that x¡=0 and on the target of the branch it would assume that indeed x¿0 holds.

Basically, the representations of dataflow are sets of program entities such as variables or expressions satisfying the given property. The classical dataflow analysis [1](e.g., live-variable, partial-redundancy elimination) implements these sets by bit-vectors. However, the new dataflow analysis problems [25, 51](e.g., linear constraint, affine relation) may need extra information about value of variables, hence these sets must be implemented by non-bit-vectors.

Dataflow analysis is used to: (1) determining the semantic validity of a program (e.g., type correctness); (2) understanding the behavior of a program for debugging, verification, testing [25, 51]; (3) transforming a program for optimized the program for space, time, or power consumption [24].

3.1.2 Model Checking

Model checking, proposed independently by E.M. Clarke and E.A. Emerson (USA) and J.Sifakis (France) in 1980 [7], is an automatic verification technique for finite state concurrent systems. The model checking problem is:

> Given a program (system model) M and a correctness specification S, determine whether or not the behavior of M satisfies the specification B?.

Applying model checking to a system consists of two main tasks:

Specifying the properties that system should have, for example, deadlock, divergence or deadlock. The specification is usually given in some logical formalism. For hardware and software systems, it is common to use temporal logic [12], which can assert how the behavior of the system evolves over time, such as CTL (computation tree logic) and LTL (linear time logic). An important issue in specification is completeness. Model checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy.

Construct a formal model for the system. In many cases, this is simply a compilation task. In other cases, the modeling of a design may require the use of abstraction to eliminate irrelevant of unimportant details. For example, when modeling programs for checking ORE, we it is useful to consider numerical variables, rather than string variables.

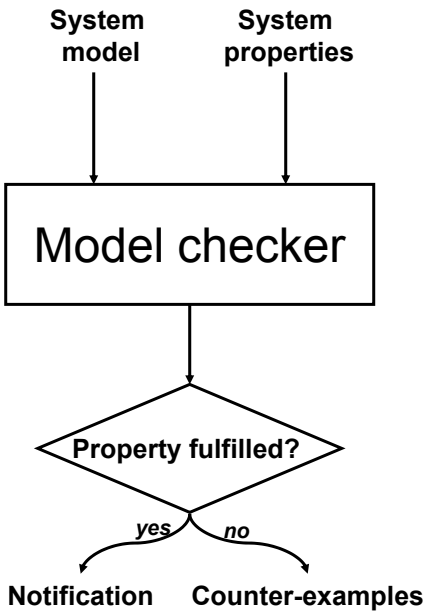


Figure 3.1: Model checker structure

A model checker (Figure 3.1) then takes the system model and system properties as the inputs. Model checker will check whether the properties are fulfilled or not? If the answer is yes then the system satisfies its properties. If the answer is no, the system violates its properties. In practice, the model checker can procedure the counterexamples for debugging purposes.

Model checking has several advantages over others verification techniques (e.g., automated theorem proving). That is, the user of a Model checker does not need to construct the correctness proof. In particular, user only need to enter a description for a system or program to be verified and the specification to be checked. The checking process is automatic. Further, model checking is fast compared to other methods such as the use of a proof checker, which may require months of the user's time working in interactive mode. However, one problem of model checking is that a counterexample can also result from incorrect modeling of the system or from an incorrect specification (often called a false negative). The counterexamples can also be useful in identifying and fixing these two problems. Another problem is that the verification task will fail to terminate normally, due to the size of the model, which is too large to fit into the computer memory. In this case, it may be necessary to redo the verification after changing some of the parameters of the model checker or by adjusting the model (e.g., using additional abstractions). Model checking for finite state systems has been successfully implemented in automatic tools such as SPIN [1], SMV/NuSMV [2].

[1] http://www.spinroot.com/
[2] http://www.cs.cmu.edu/modelchecker/code.html

3.1.3 Dataflow Analysis as Model Checking and Abstraction

As we known, the classical dataflow analysis can be transformed to model checking problem by creating suitable abstractions of the programs as models and expressing desired properties in terms of temporal formulae (e.g., CTL formulae) [34].

Modeling program Given a program, the standard program model includes:

- *program states* are the program locals (e.g., program point),
- *actions* are the elementary statements and expressions, and
- *transitions* are defined by the small steps and are labeled with corresponding primitive statements/expressions.

The programs can be modeled as labeled transition systems.

Definition 12 *A **label transition system** $\mathcal{P}$ is a tuple (S, A, Δ) in which*

- *S is a finite set of nodes or program states,*
- *A is a set of actions, modeling elementary statements,*
- *$\Delta \subseteq S \times A \times S$ is a set of labeled transitions, which modeling the flow of control, and*

Example 13 *For the program in Figure 1.2, the transition system is $P1 = (S, A, \Delta)$ with:*

- *$S = \{st1, st2, \cdots, st5\}$*
- *A is the set of statements of the program*
- *Δ is defined as figure 3.2*

Next, for dataflow analysis problem, the states must be labeled with sets of atomic propositions which describe properties of states:

Definition 13 *A **labeled state transition system** $\mathcal{P}_l$ is a triplet(P, B, λ) in which*

- *P is a label transition system (S, Δ, s_0),*
- *B is a set of atomic propositions, and*
- *λ is a function $\lambda : S \to 2^B$ that labels states with subsets of B.*

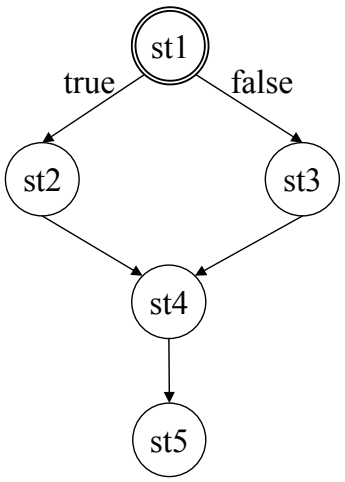


Figure 3.2: Transition system of program in Figure 1.2

Given a labeled transition system, one can define the corresponding labeled state transition system by using a lattice of entities of observation D. Then, a monotonic transfer function will be defined for each action $a \in A$ $f_a : D \rightarrow D,\ a \in A$. Lastly, each program state $p \in S$ will be labeled by $val_p = \bigsqcup\{f_a(val_q) |\ (p, a, q) \in \Delta\}$.

Example 14 *Let us consider live-variables analysis for the program in Figure 1.2. The labeled transition system is given in Example 13. The corresponding labeled state transition system will be given as follows:*

- $D = 2^{Var}$*, where* $Var = \{x, y, rst\}$ *is the collection of the program's variables;*
- $f_a(s) = Used_a \cup (notModified_a \cap s)$*, where* $Used_a$ *defines those variables referenced in action a,a nd* $notModified$ *defines those variables that are not modified in a.*

Abstraction in Dataflow Analysis Data abstraction is probably the most important technique for reducing the state explosion theorem. Data abstraction is based on the observation that the specifications of the programs that include data paths usually involve fairly simple relationships among the data values in the program. The abstraction is usually specified by giving a mapping between the actual data values of variables in the programs and a small set of abstract data values.

Definition 14 *For labeled transition system* $P_l = (P, B, \lambda)$ *and the abstract function* $f_A : B \rightarrow B'$ *such that* $|B'| < B$*, the abstract transition system of* P_l *is* $P_a = (P, B', \lambda')$ *where* $\lambda' : S \rightarrow 2^{B'}$

The abstract model is often much smaller than the actual model, and as a result, it is usually much simpler to verify properties at the abstract level.

Example 15 *For the program in Figure 1.2, assume that the initial values of variable* x, y *lie in* $\{-1, 1, 3\}$*. We would like to check whether the result of* rst *is even or odd.*

The transition system $P1$ *is already given in Example 13. Directly, we can set the states of the transition system* $P1$ *in example are all possible values of* x, y, rst*. Hence,* $B = \{-1, 0, 1, 2, 3, 4, 9, ...\}$*. However, we can safety abstract* B *to* $B' = \{even, odd\}$ *where* $f_A(x) = even$ *if* x *div 2,* $f_A(x) = odd$ *otherwise. By this, the abstract model will become simpler than the concrete model.*

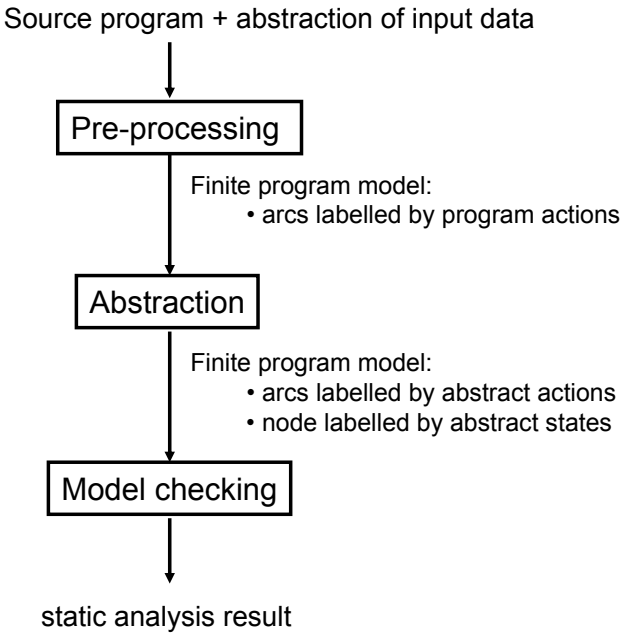


Figure 3.3: Dataflow analysis as model checking and abstraction

To apply model checker as the engine for dataflow analysis, a third component, abstraction, must be used. First, from a operational semantics definition and a program, one constructs a program model, which is a state-transition system that encodes one (or many, or all) of the program's executions. Second, one might abstract upon the program model, reducing the detail of information in the model's nodes and arcs. Finally, one analyzes the model for the properties, etc (Figure 3.3).

3.2 Dataflow Analysis as Weighted Model Checking Problem

3.2.1 Weighted Model Checking

Weighted model checking (WMC) 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. In one hand, the computation of fixpoint often require widening operation that causes over approximation too much (e.g., approximate value of variable to ∞). In other hand, the most important requirement of ORE analysis is precision. Thus, in order to avoiding widening operation, we restrict the underlying pushdown system to a finite state transition system. By this, the weighted pushdown model checking becomes weighted model checking.

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

Definition 15 *An **idempotent semiring** is a quintuple $(D, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$, where $\boldsymbol{0}, \boldsymbol{1} \in D$ 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 $\boldsymbol{0}$,*
- *$(D, \otimes)$ is a monoid with the unit $\boldsymbol{1}$,*
- *$\otimes$ 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., $a \otimes \boldsymbol{0} = \boldsymbol{0} \otimes a = \boldsymbol{0}$.*

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

- **0** stands for interruption of dataflow,
- **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 16 *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 17 *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 18 *An idempotent semiring is* ***bounded*** *if there are no infinite descending chains wrt $\sqsubseteq$, where $a \sqsubseteq b$ if, and only if, $a \oplus b = a$.*

Data Abstraction and Weight Domain

For the purpose of dataflow analysis, the weights capture the relationships between the values of variables before and after the statements. Thus, the weight domain can be treated as a set of abstract transformers.

More formally, let $V, \mathbb{V}$ be the set of variables and the set of their corresponding values, respectively. The abstract function is $f_A : \mathbb{V} \to \mathbb{V}_A$ where $\mathbb{V}_A$ is abstract domain of $\mathbb{V}$. Hence, the weight domain D is the set of functions $f : \mathbb{V}_A \to \mathbb{V}_A$ which stands for the transformations from abstract values of variables before statements to the abstract values of variables after statements.

3.2.2 Dataflow Analysis as Weighted Model Checking and Abstraction

Solving dataflow analysis problem based on weighted model checking basically includes three main steps (Figure 3.4). First, the pre-processing step obtains the transition system and input domain of source program. Next, by abstracting the input domain, we obtain the abstract domain and the corresponding weighted transition system. This is the input of Step 3, weighted model checking. Finally, we obtain the analysis result.

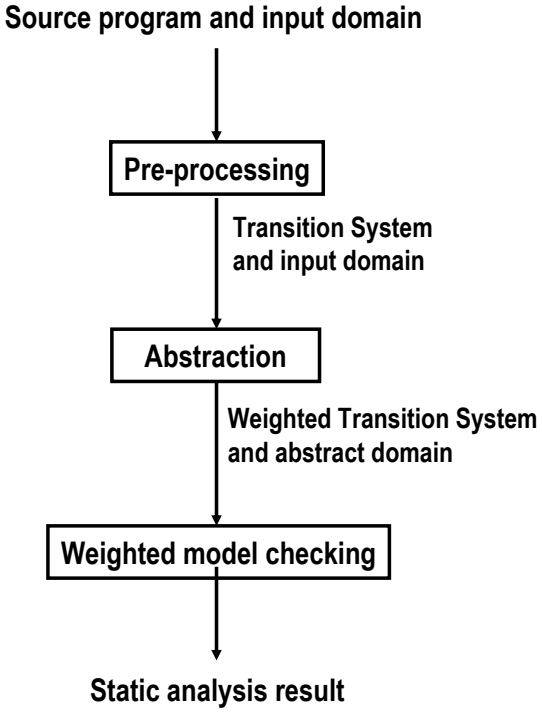


Figure 3.4: Dataflow analysis as weighted model checking

For dataflow analysis, common actions are extending the weights along one path, and composing the weights of paths. Each element of a bounded idempotent semiring is regarded as follows:

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

The target weighted transition system consists of

- State: is a pair of program location and abstract environment,

- The initial state s_0: is the program entry, and

Note that for if statement "if $x \circ y$ then s", the abstraction of the condition $x \circ y$ affects the analysis; its dataflow is interrupted (by the weight $\mathbf{0}$) when $(x \circ y)$ is evaluated to **false**.

3.2.3 On-the-fly Weight Creation for an Acyclic Model

Because of the requirement of dataflow properties, the abstract domain is sometimes still infinitely many. Thus, designing the weight domain that satisfy the descending chain condition (boundedness) is difficult. For example, the ORE analysis abstracts OREs as intervals, which is infinite domain.

We now create weight domain for a subclass of program model called "acyclic model", in which weights can be designed by on-the-fly manner. Let val_0 be the abstract value of input domain, val_0 will be the parameter for the on-the-fly weight domain generation. We first introduce the augmented weight domain to associate an input abstract environment to each weight. "_" means any input.

Definition 19 *The* augmented weight domain $\mathcal{S}^+ = (D^+, \oplus, \otimes, \boldsymbol{0}^+, \boldsymbol{1}^+)$ *consists of* $D^+ = \{(W, w) \mid W \in AbsEnv, w \in D\}$, $\boldsymbol{0}^+ = (\_, \boldsymbol{0})$, $\boldsymbol{1}^+ = (\_, \boldsymbol{1})$, *and*

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

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

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 $\mathcal{S}^+_{\mathcal{P},val_0}$ for a transition system $\mathcal{P}$ and $Val_0 \in \mathbb{V}_A$. The intuition is, starting from the initial abstract environment val_0, only reachable instances of weights are computed in on-the-fly manner.

Definition 20 *For a transition system* $\mathcal{P}$ *and* $val_0 \in \mathbb{V}_A$, *the weight domain* $\mathcal{S}^+_{\mathcal{P},val_0} = (D^+_{\mathcal{P},val_0}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ *is a sub semiring of* $\mathcal{S}^\times$ *with* $D^+_{\mathcal{P},val_0} \subseteq D^+$. $D^+_{\mathcal{P},val_0}$ *is given by*

$$\left\{ (W, w) \;\middle|\; \begin{array}{l} \exists \sigma, \sigma' \in \Delta^* \;\; \exists c, c' \in P.\;\; s_0 \Rightarrow^{\sigma} c \Rightarrow^{\sigma'} c' \\ \wedge W = v(\sigma)(val_0) \wedge w = v(\sigma') \end{array} \right\}$$

Chapter 4

Interval Arithmetics in ORE Propagation

In general, we do not know the exact values of variables in the program, only their ranges. Hence, estimating ORE can be solved by propagating fixed point ranges and round off error ranges of variables. In order to estimate OREs of operations, there are two known range representations: classical interval [47] and affine interval [61, 62]. In this chapter, we firstly describe these two methods in detail in Sections 4.1 and 4.2. Then, motivated by the disadvantages of these two interval methods, we propose two novel range representation methods, called "extended affine interval" and "positive-noise affine interval", in Section 4.3 and Section 4.4, respectively. Lastly, we represent how to implement these intervals on computers using floating point type in Section 4.5.

4.1 Classical Interval

Classical interval (CI) was introduced in the 1960s by Moore [47] as an approach to putting bounds on rounding errors in mathematical computations. In CI, each quantity is represented by the set of all possible values. Formally, CI is defined as follows:

Definition 21 *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 $\overline{R}$.*

The result of CI arithmetic is also a CI that binds all possible results. In particular, CI arithmetic is evaluated as follows:

Definition 22 ***CI arithmetic** consists of operations $\{\overline{+}, \overline{-}, \overline{\times}, \overline{\div}\}$ on pairs of CIs*

defined below:

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

The following example demonstrates how to compute CI operations:

Example 16 *For* $x \in \overline{x} = [-1, 3],\ y \in \overline{y} = [-6, 10]$. *Let us compute the bound of* $z = x \circ y$ *(*$\circ \in \{+,\ -,\ \times,\ \div\}$*) by using CI:*

- *Addition* $z = x + y$*:*

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

  Hence, we can conclude $z \in [-7, 13]$.

- *Substraction* $z = x - y$*:*

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

  Hence, we can conclude $z \in [-11, 9]$.

- *Multiplication* $z = x \times y$*:*

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

  Hence, we can conclude $z \in [-18, 30]$.

- *Division* $z = x \div y$*:*

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

  Because $0 \in [-6, 10]$*, hence we cannot compute the bound of* z*, in this case a "divide for zero" warning might be displayed.*

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:

- $\overline{x}_1\overline{x}_2 = \overline{x}_1 \ \overline{\times}\ \overline{x}_2$, $c\overline{x} = \overline{x}c = \overline{x} \ \overline{\times}\ [c,\ c]$,
- $c \ \circ\ \overline{x} = [c, c] \ \circ\ \overline{[x]}$, $\overline{x} \ \circ\ c = \overline{x} \ \circ\ [c, c]$, 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 their corresponding quantities are dependent. The next example illustrates such a problem.

Example 17 *Let $x \in \overline{x} = [-1, 3]$. It is easy to see that:*

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

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

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

4.2 Affine Interval

Affine interval (AI) was introduced by Stolfi [61, 62] as a model for self-validated numerical analysis. It was proposed to address the "error explosion" problem in conventional CI. Unlike CI, in AI, the quantities 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 23 *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 (i.e., addition, subtraction) are straightforward operations on AIs. However, the results of nonlinear operations (i.e., multiplication, division) are not AI forms. Hence, we need to approximate the nonlinear parts of the results by introducing new noise symbols.

Definition 24 ***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:*

$$\begin{aligned}
\ddot{x}\ \ddot{+}\ \ddot{y} &= (x_0 + y_0)\ +\ \textstyle\sum_{i=1}^{n}(x_i\ +\ y_i)\varepsilon_i \\
\ddot{x}\ \ddot{-}\ \ddot{y} &= (x_0\ -\ y_0)\ +\ \textstyle\sum_{i=1}^{n}(x_i\ -\ y_i)\varepsilon_i \\
\ddot{x}\ \ddot{\times}\ \ddot{y} &= x_0y_0\ +\ \textstyle\sum_{i=1}^{n}(x_0y_i\ +\ x_iy_0)\varepsilon_i\ +\ B\varepsilon_{n+1} \\
\ddot{x}\ \ddot{\div}\ \ddot{y} &= \ddot{x}\ \ddot{\times}\ (\tfrac{1}{\ddot{y}}), \textit{if } 0 \notin [x_0\ -\ \textstyle\sum_i^n |x_i|, x_0\ +\ \sum_i^n |x_i|]
\end{aligned}$$

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 [61].*

Remark 5 *An easy approximation of* B *is* $(\sum_{i=1}^{n} |x_i|)(\sum_{i=1}^{n} |y_i|)$

Chebyshev approximation

In general, Chebyshev approximation aims to minimize the maximum absolute error. Specifically, let $\mathcal{F}$ be some space of functions, (polynomials, affine forms, etc.). An element of $\mathcal{F}$ that minimizes the maximum absolute difference from a given function f over some specified domain Ω is known as a Chebyshev (or minimax) $\mathcal{F}$-approximation to f over Ω.

In particular, for univariate functions, the minimax affine approximation is characterized by the following property:

Theorem 1 *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:

Theorem 2 *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 the AI projection of $\ddot{y}$, $\overline{y}$, includes zero, a "division by zero" might be displays. 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 $\frac{1}{\ddot{y}}$ (Figure 4.1) 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.

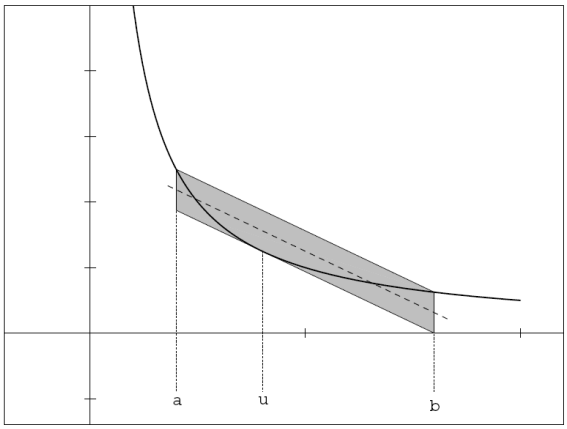


Figure 4.1: Chebyshev approximation for $\frac{1}{\ddot{y}}$

Conversion between CI and AI

Standard range representation is a 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 about source of uncertainty. This is called *AI projection*.

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

Example 18 *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 $z = x \circ y$ ($\circ \in \{+,\ -,\ \times,\ \div\}$) by using AI:

- *Addition $z = x + y$:*

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

  The AI projection of $\ddot{z}$ is $[3 - 2 - 8, 3 + 2 + 8] = [-7, 13]$. Hence, we can conclude $z \in [-7, 13]$.

- *Substraction $z = x - y$:*

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

  The AI projection of $\ddot{z}$ is $[-1-2-8, -1+2+8] = [-11, 9]$. Hence, we can conclude that $z \in [-11, 9]$.

- *Multiplication $z = x \times y$:*

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

  where ε_1 is new noise symbol standing for $\varepsilon_x\varepsilon_y$.

  The AI projection of $\ddot{z}$ is $[2 - 4 - 8 - 16, 2 + 4 + 8 + 16] = [-26, 30]$.

  Hence, we can conclude that $z \in [-26, 30]$.

- *Division $z = x \div y$:*

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

  Because $0 \in [-6, 10]$, hence we cannot compute the bound of z, in this case a "divide for zero" warning might be displayed.

AI is more precise than CI for linear operations, as shown in the next example.

Example 19 *For $x \in \overline{x} = [-1, 3]$, let us consider how to compute the bound of $z = x - x$ by using CI and AI.*

- *CI substraction:*

$$\overline{z} = \overline{x} \stackrel{-}{-} \overline{x} = [-4, 4]$$

  We hence conclude $z \in [-4, 4]$.

- *AI substraction: The AI coercion of $\overline{x}$ is $\ddot{x} = 1 + 2\varepsilon_x$. Hence:*

$$\ddot{z} = \ddot{x} \stackrel{\cdot\cdot}{-} \ddot{x} = 0$$

  We hence conclude $z = 0$. This result is the correct result of the subtraction $(x - x)$.

However, in AI arithmetic, each time we perform a nonlinear operation, we introduce a new noise symbol, which is problematic for a program with a large number of nonlinear operations.

4.3 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 [27], 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 over approximation. This is called an *extended affine interval* (EAI), which also avoids introduction of new noise symbols for nonlinear operations.

Definition 25 *An **extended affine interval** of x is a formula*

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

with $x \in \overline{x}_0 \stackrel{-}{+} \sum_{k=1}^{n} \overline{x}_k [-1, 1]$, 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 are guaranteed to be EAIs by approximating nonlinear parts. For example, let us consider the multiplication of two EAIs. Let $\widehat{x} = \overline{x}_0 + \sum_{i=1}^{n} \overline{x}_i \varepsilon_i$, $\widehat{y} = \overline{y}_0 + \sum_{i=1}^{n} \overline{y}_i \varepsilon_i$. 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: $\hat{x} \mathbin{\hat{\times}} \hat{y} = (\overline{x}_0 + \sum_{i=1}^{n} \overline{x}_i\varepsilon_i) \mathbin{\hat{\times}} (\overline{y}_0 + \sum_{i=1}^{n} \overline{y}_i\varepsilon_i)$ $= \overline{x}_0\overline{y}_0 + \sum_{i=1}^{n}(\overline{x}_0\overline{y}_i \mathbin{\overline{+}} \overline{x}_i\overline{y}_0 \mathbin{\overline{+}} \overline{x}_iB)\varepsilon_i$, where $B = \sum_{i=1}^{n} \overline{y}_i\varepsilon_i$. An easy approximation of B is $\sum_{k=1}^{n} \overline{y}_k[-1,1]$. Formally, EAI arithmetic is defined as follows:

Definition 26 ***EAI arithmetic*** *consists of operations* $\{\hat{+}, \hat{-}, \hat{\times}, \hat{\div}\}$ *on pairs of EAIs.*

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

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

where:

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

and $\frac{1}{\hat{y}}$ *is computed by Chebyshev approximation [61].*

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 6 *The over approximation* B *may conceal some noise symbols. If we are sensitive to this matter,* B *can be modified as:*

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

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

Conversion between CI and EAI

Standard range representation is a CI. To apply EAI, the conversion between them are needed.

- *CI to EAI*: Given a CI $\overline{x} = [l,\ h]$, a corresponding EAI is $\hat{x} = \frac{l+h}{2} + \frac{h-l}{2}\varepsilon_k$. 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 $\hat{x} = \overline{x}_0 + \Sigma_{i=1}^{n}\overline{x}_i\varepsilon_i$ 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 about source of uncertainty. This is called *EAI projection*.

The following example demonstrates how to propagate the ranges using EAI:

Example 20 *For* $x \in \overline{x} = [-1, 3],\ y \in \overline{y} = [-6, 10]$. *The corresponding EAI coercions of* $\overline{x},\ \overline{y}$ *are:*

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

Let us compute the bound of $z = x \circ y$ *(*$\circ \in \{+,\ -,\ \times,\ \div\}$*) by using EAI:*

- *Addition* $z = x + y$*:*

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

  The EAI projection of $\hat{z}$ *is* $3 \overline{+} 2[-1, 1] \overline{+} 8[-1, 1]\ =\ [-7, 13]$.

  Hence, we can conclude that $z \in [-7, 13]$.

- *Substraction* $z = x - y$*:*

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

  The AI projection of $\ddot{z}$ *is* $-1 \overline{+} 2[-1, 1] \overline{-} 8[-1, 1]\ =\ [-11, 9]$.

  Hence, we can conclude that $z \in [-11, 9]$.

- *Multiplication* $z = x \times y$*:*

$$\begin{aligned}\hat{z} &= \hat{x}\ \hat{\times}\ \hat{y}\\ &= (1\ +\ 2\varepsilon_x)\ \hat{\times}\ (2\ +\ 8\varepsilon_y)\\ &= 2\ +\ 4\varepsilon_x\ +\ 8\varepsilon_y\ +\ B\end{aligned}$$

  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,* $\hat{z} = 2\ +\ 4\varepsilon_x\ +\ [-8, 24]\varepsilon_y$

  The AI projection of $\ddot{z}$ *is* $2 \overline{+} 4[-1, 1] \overline{+} [-8, 24]\overline{\times}[-1, 1]\ =\ [-26, 30]$.

  Hence, we can conclude that $z \in [-26, 30]$

- *Division* $z = x \div y$*:*

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

Because $0 \in [-6, 10]$*, hence we cannot compute the bound of* z*, in this case a "divide for zero" warning might be displayed.*

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 some nonlinear parts. Let us consider the example below:

Example 21 *Let* $z = x \times x; t = z - z$ *and* $x \in [-1, 31]$.

The bound of t *is computed based on AI and EAI arithmetics as follows:*

- *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:* $\hat{x} = 1\overline{2}\varepsilon_x,\ \hat{z} = 1\overline{+}[0, 8]\varepsilon_x$.

  $\hat{t} = \hat{z}\hat{-}\hat{z} = [-8, 8]\varepsilon_x$

The bound of $\ddot{z}$*, [0,0], lies within the bound of* $\hat{z}$*, [-8,8]. So, AI arithmetic is more precise in this case. However, if we compute the bound of* $t = x \times x - x \times x$ *without reusing the multiplication* $x \times x$*, then both AI arithmetic and EAI arithmetic return the same bound.* $\ddot{t}$, $\hat{t}$ can be computed in a similar way. To keep things simple, we omit the details of these computations here.

4.4 Positive-noise Affine Interval

In this section, we propose another way to "extend" AI which tries to reduce the over approximation in the nonlinear operations.

We start by an example:

Example 22 *Let us consider the multiplication:* $x = \varepsilon_k \times \varepsilon_k$.

It easily sees that $x \in [0, 1]$ *for all* $\varepsilon_k \in [-1, 1]$*. However, by applying AI multiplication and EAI multiplication, we get:*

- *AI multiplication* $\ddot{x} = \varepsilon_k \ddot{\times} \varepsilon_k$*. We have:*

  $\ddot{x} = \varepsilon_{n+1}$*, where* $B = 1$ *and* ε_{n+1} *is new noise symbol.*

  Hence, $\ddot{x} \in [-1, 1]$ *(instead of* $[0, 1]$*) and we already lose information about* ε_k.

- *EAI multiplication* $\hat{x} = \varepsilon_k \hat{\times} \varepsilon_k$*. We have:*

  $\hat{x} = [-1, 1]\varepsilon_k$.

  Hence, $\hat{x} \in [-1, 1]$ *(instead of* $[0, 1]$*).*

Therefore, both AI and EAI are over approximate too much. Further, If we apply Chebyshev approximation, the result will be more precise: $\hat{x} = [0, 1]$. *However, this result still not keep information about* ε_k.

Note that, if $\varepsilon_k \in [0, 1]$ *Chebyshev approximation will return* $\hat{x} = \varepsilon_k + [\frac{-1}{4}, 0]$. *This result still keep information about* ε_k.

In AI arithmetic and EAI arithmetic, the over approximations often occur in non-linear operations, especially when multiplying of operands which are same noise symbol. The above example give us a new idea to extend AI for reducing over approximation in multiplication. In particular, the noise symbol $\varepsilon_i \in [0, 1]$ (instead of $\varepsilon_i \in [-1, 1]$) and the multiplication between same noise symbol will be refined by applying Chebyshev approximation.

Definition 27 *An* ***Positive-noise affine interval*** *of* x *is a formula*

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

with $x \in \overline{x}_0 - + \sum_{k=1}^{n} \overline{x}_k [0, 1]$, *where* $\varepsilon_i \in [0, 1]$ *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 positive-noise affine intervals is denoted by* $\widetilde{R}$.

The linear operations of positive-noise affine interval (PAI) arithmetic are designed similarly to those of EAI arithmetic. Unlike EAI, the PAI multiplication approximate the multiplication between same noise symbol by apply Chebyshev approximation. In particular, PAI arithmetic is defined formally as follows:

Definition 28 ***PAI arithmetic*** *consists of operations* $\{\widetilde{+}, \widetilde{-}, \widetilde{\times}, \widetilde{\div}\}$ *on pairs of PAIs. Let* $\widetilde{x} = \overline{x}_0 + \sum_{i=1}^{n} \overline{x}_i \varepsilon_i$, $\widetilde{y} = \overline{y}_0 + \sum_{i=1}^{n} \overline{y}_i \varepsilon_i$, $\overline{X_i} = \sum_{k=1, k \neq i}^{n} (\overline{x}_k\ [0, 1])$, *and* $\overline{Y_i} = \sum_{k=1, k \neq i}^{n} (\overline{y}_k\ [0, 1])$. *Then,*

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

where:

$$B = \begin{cases} \sum_{i=1}^{n} (\overline{x}_i \varepsilon_i \overline{Y_i} + \overline{x}_i \overline{y}_i \varepsilon_i + [-1/4, 0]) & \textit{if } \overline{Y_0} \subseteq \overline{X_0} \\ \sum_{i=1}^{n} (\overline{X}_i \overline{y}_i \varepsilon_i + \overline{x}_i \overline{y}_i \varepsilon_i + [-1/4, 0]) & \textit{otherwise} \end{cases}$$

and $\frac{1}{\widetilde{y}}$ *is computed by Chebyshev approximation [61].*

Remark 7 *Similar to EAI arithmetic, the over approximation B may conceal some noise symbols. If we are sensitive to this matter, B can be modified as:*

$$B = \alpha(\sum_{i=1}^{n} \overline{x}_i \overline{Y}_i \varepsilon_i) \ + \ \beta(\sum_{i=1}^{n} \overline{X}_i \overline{y}_i \varepsilon_i) \ + \ \overline{x}_i \overline{y}_i \varepsilon_i \ + \ [-1/4, 0]$$

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

Conversion between CI and PAI

- *CI to PAI*: Given a CI $\overline{x} = [l, \ h]$, an PAI is $\widetilde{x} = l \ + \ (h - l)\varepsilon_k$. Under valuations of a noise symbol ε_k to $[0, 1]$, they represent the same range. This is called *PAI coercion*.
- *PAI to CI*: An PAI $\widetilde{x} = \overline{x}_0 \ + \ \Sigma_{i=1}^{n} \overline{x}_i \varepsilon_i$ is projected to a CI $\overline{x} = \overline{x}_0 \ \overline{+} \ \Sigma_{i=1}^{n} \overline{x}_i [0, \ 1]$. Replacement of a noise symbol ε_k to $[0, 1]$ loses information about source of uncertainty. This is called *PAI projection*.

Like EAI, PAI does not introduce new noise symbols for nonlinear operation. Further, it is designed to reduce the over approximation in multiplication which is often appear in ORE analysis. PAI is sometimes more precise than both EAI and AI as following examples:

Example 23 *Let* $x \in [1, \ 3]$ *and* $y \in [-4, \ 2]$, *we want to find the bound of* $z = x \times x \ + \ x \times y \ - \ y$.

We will find the bound of z by applying AI, EAI and PAI arithmetics.

1. ***AI arithmetic***

   Using AI coercion, we get: $\ddot{x} = 2 \ + \ \varepsilon_1$ *and* $\ddot{y} = -3 \ + \ \varepsilon_2$ *for* $\varepsilon_1, \ \varepsilon_2 \in [-1, 1]$.

   We have:

$$\begin{array}{lll} \ddot{x} \ \ddot{\times} \ \ddot{x} = 4 \ + \ 4\varepsilon_1 \ + \ \varepsilon_3 & & \text{where } \varepsilon_3 \text{ is new noise symbol standing for } \varepsilon_1 \ \times \ \varepsilon_1 \\ \ddot{x} \ \ddot{\times} \ \ddot{y} = -6 \ - \ 3\varepsilon_1 \ + \ 2\varepsilon_2 \ + \ \varepsilon_4 & & \text{where } \varepsilon_4 \text{ is new noise symbol standing for } \varepsilon_1 \ \times \ \varepsilon_2 \\ \ddot{z} \quad = \ddot{x} \ \ddot{\times} \ \ddot{x} \ \ddot{+} \ \ddot{x} \ \ddot{\times} \ \ddot{y} \ \ddot{-} \ \ddot{y} & & \\ \quad = 1 \ + \ \varepsilon_1 \ + \ \varepsilon_2 \ + \ \varepsilon_3 \ + \ \varepsilon_4 & & \end{array}$$

   Applying AI projection, we get the bound of z is $[-3, 5]$.

2. ***EAI arithmetic***

   Using EAI coercion, we get: $\hat{x} = 2 \ + \ \varepsilon_1$ *and* $\hat{y} = -3 \ + \ \varepsilon_2$ *for* $\varepsilon_1, \ \varepsilon_2 \in [-1, 1]$.

   We have:

$$\begin{aligned}
\hat{x} \mathbin{\hat{\times}} \hat{x} &= 4 + 4\varepsilon_1 + [-1,1]\varepsilon_1 \\
\hat{x} \mathbin{\hat{\times}} \hat{y} &= -6 - 3\varepsilon_1 + 2\varepsilon_2 + [-1,1]\varepsilon_1 \\
\hat{z} &= \hat{x} \mathbin{\hat{\times}} \hat{x} \mathbin{\hat{+}} \hat{x} \mathbin{\hat{\times}} \hat{y} \mathbin{\hat{-}} \hat{y} \\
&= 1 + [-1,3]\varepsilon_1 + \varepsilon_2
\end{aligned}$$

Applying EAI projection, we get the bound of z is $[-3,5]$.

3. ***PAI arithmetic***

   Using PAI coercion, we get: $\widetilde{x} = 1 + 2\varepsilon_1$ and $\widetilde{y} = -4 + 2\varepsilon_2$ for $\varepsilon_1,\ \varepsilon_2 \in [0,1]$.

   We have:

$$\begin{aligned}
\widetilde{x} \mathbin{\widetilde{\times}} \widetilde{x} &= 1 + 8\varepsilon_1 + [-1,0] \\
\widetilde{x} \mathbin{\widetilde{\times}} \widetilde{y} &= -4 - 8\varepsilon_1 + 2\varepsilon_2 + 4[0,1]\varepsilon_1 \\
\widetilde{z} &= \widetilde{x} \mathbin{\widetilde{\times}} \widetilde{x} \mathbin{\widetilde{+}} \widetilde{x} \mathbin{\widetilde{\times}} \widetilde{y} \mathbin{\widetilde{-}} \widetilde{y} \\
&= [0,1] + [0,4]\varepsilon_1
\end{aligned}$$

   Applying PAI projection, we get the bound of z is $[0,5]$.

In conclusion, PAI returns more precise bound than both AI and EAI do.

4.5 Interval Representations by Floating point Numbers

The above intervals are defined by using real numbers. As we known, the computers have to represent real numbers in finite form. To implement above interval arithmetics in the computer, we need to ensure that the during the computation, the OREs do not affect the correctness of the result or the floating point intervals have to bound the real intervals.

As pointed in [61, 62], the interval representations themselves are affected by OREs, since boundaries and coefficients are represented by floating point numbers. We will briefly overview how to over approximate 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 29 *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 over approximation, 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 symbol.

Definition 30 *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 31 *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*

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

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 coefficient 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.

Chapter 5

Abstraction for ORE Problem

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. In this chapter, we will show three kinds of abstractions based on CI, AI, and EAI range representations.

5.1 CI Abstraction for ORE Problem

For estimation of propagations of REs based on CI, we apply CI arithmetic (Definition 22) to Float64-to-fixed16 ORE arithmetic (Definition 11).

Definition 32 *The set* $\overline{\Phi}$ *of* ***CI abstract numbers*** *is:*

$$\overline{\Phi} = \{(fxp, rdf) | fxp, rdf \in \overline{R}\}$$

CI abstract arithmetic *consists of* $\{\overline{\boxplus},\ \overline{\boxminus},\ \overline{\boxtimes},\ \overline{\boxdiv}\}$ *on pairs of CI abstract numbers. Let* $(\overline{x}_f,\ \overline{x}_r),\ (\overline{y}_f,\ \overline{y}_r) \in \overline{\Phi}$ *be CI abstract numbers. The CI abstract operators are defined below:*

$$(\overline{x}_f,\ \overline{x}_r)\ \overline{\boxplus}\ (\overline{y}_f,\ \overline{y}_r) = (\overline{x}_f \ \overline{+}\ \overline{y}_f,\ \overline{x}_r \ \overline{+}\ \overline{y}_r \ \overline{+}\ [-\delta_+, \delta_+])$$
$$(\overline{x}_f,\ \overline{x}_r)\ \overline{\boxminus}\ (\overline{y}_f,\ \overline{y}_r) = (\overline{x}_f \ \overline{-}\ \overline{y}_f,\ \overline{x}_r \ \overline{-}\ \overline{y}_r \ \overline{+}\ [-\delta_-, \delta_-])$$
$$(\overline{x}_f,\ \overline{x}_r)\ \overline{\boxtimes}\ (\overline{y}_f,\ \overline{y}_r) = (\overline{x}_f \ \overline{\times}\ \overline{y}_f,\ \overline{x}_r \ \overline{\times}\ \overline{y}_f \ \overline{+}\ \overline{x}_f \ \overline{\times}\ \overline{y}_r \ \overline{+}\ \overline{x}_r \ \overline{\times}\ \overline{y}_r \ \overline{+}\ [-\delta_\times, \delta_\times])$$
$$(\overline{x}_f,\ \overline{x}_r)\ \overline{\boxdiv}\ (\overline{y}_f,\ \overline{y}_r) = (\overline{x}_f \ \overline{\div}\ \overline{y}_f,\ (\overline{x}_f \ \overline{+}\ \overline{x}_r) \ \overline{\div}\ (\overline{y}_f \ \overline{+}\ \overline{y}_r) \ \overline{-}\ \overline{x}_f \ \overline{\div}\ \overline{y}_f \ \overline{+}\ [-\delta_\div, \delta_\div])$$

where $\delta_+, \delta_-, \delta_\times, \delta_\div$ *is given in Equation 2.2.*

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

Example 24 *The CI abstract numbers* $(\overline{x}_f, \overline{x}_r)$, $(\overline{y}_f, \overline{y}_r)$ *for variables* x, y, *respectively, in Example 3 are*

$$\begin{cases} \overline{x}_f = [-1,\ 3] \\ \overline{x}_r = [-2^{-5},\ 2^{-5}] \\ \overline{y}_f = [-10,\ 10] \\ \overline{y}_r = [-2^{-5},\ 2^{-5}] \end{cases}$$

since $fp = 5$, *the REs of* x *and* y *are in* $[-2^{-fp-1}, 2^{-fp-1}] = [-2^{-5}, 2^{-5}]$ $(= [-0.03125, 0.03125])$.

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

$$\begin{aligned} \overline{\texttt{rst}}_f &= \overline{x}_f\ \overline{\times}\ \overline{x}_f \\ &= ([-1,3]\ \overline{\times}\ [-1,3] \\ &= [-3,9] \end{aligned}$$

$$\begin{aligned} \overline{\texttt{rst}}_r &= 2\ \overline{\times}\ \overline{x}_f\ \overline{\times}\ \overline{x}_r\ \overline{+}\ \overline{x}_r\ \overline{\times}\ \overline{x}_r\ \overline{+}\ \overline{\delta} \\ &= [-0.219726563, 0.219726563] \end{aligned}$$

CI abstract comparison operators

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 operators are defined by using Float-to-Fixed comparison operators as follows:

Replacing (x_f, x_r), (y_f, y_r) in Float-to-fixed ORE comparison operators (Definition 8) with CI abstract numbers $(\overline{x}_f, \overline{x}_r)$, $(\overline{y}_f, \overline{y}_r)$ we obtain CI abstract comparison operators:

Definition 33 *Let* $(\overline{x}_f, \overline{x}_r)$, $(\overline{y}_f, \overline{y}_r) \in \overline{\Phi}$, *let* $\overline{x} = (\overline{x}_f\ \overline{+}\ \overline{x}_r)$, *and let* $\overline{y} = (\overline{y}_f\ \overline{+}\ \overline{y}_r)$.

$$((\overline{x}_f, \overline{x}_r)\ \overline{<}\ (\overline{y}_f, \overline{y}_r)) = \begin{cases} \textbf{\textit{true}} & \text{if } \forall u \in \overline{x}\ \forall v \in \overline{y}.u < v\ \ \wedge\ \ \forall u \in \overline{x}_f\ \forall v \in \overline{y}_f.u < v \\ \textbf{\textit{false}} & \text{if } \forall u \in \overline{x}\ \forall v \in \overline{y}.u \geqslant v\ \ \wedge\ \ \forall u \in \overline{x}_f\ \forall v \in \overline{y}_f.u \geqslant v \\ \textbf{\textit{unknown}} & \text{otherwise} \end{cases}$$

$$((\overline{x}_f, \overline{x}_r)\overline{=}(\overline{y}_f, \overline{y}_r)) = \begin{cases} \textbf{\textit{true}} \text{ if } (\forall u \in \overline{x}_f\ \forall v \in \overline{y}_f.u = v\ \ \wedge\ \ \forall u \in \overline{x}_r\ \forall v \in \overline{y}_r.u = v) \\ \textbf{\textit{false}} \text{ if } (\overline{x}_f, \overline{x}_r)\overline{<}(\overline{y}_f, \overline{y}_r)\ \ \vee\ \ (\overline{y}_f, \overline{y}_r) < (\overline{x}_f, \overline{x}_r) \\ \textbf{\textit{unknown}} \text{ otherwise} \end{cases}$$

The following example illustrates how to evaluate CI abstract comparison operator $\overline{<}$.

Example 25 *Use* $(\overline{x}_f, \overline{x}_r)$, *and* $(\overline{y}_f, \overline{y}_r)$ *as in Example 24.* $(\overline{x}_f, \overline{x}_r)\ \overline{<}\ [0,0]$ *is evaluated as follows:*

$$\begin{aligned}\overline{x}&=\overline{x}_f \mathbin{\overline{+}} \overline{x}_r\\ &=[-1,3] \mathbin{\overline{+}} [-2^{-5},2^{-5}]\\ &=[-1-2^{-5},3+2^{-5}]\end{aligned}$$

Since $0 \in [-1-2^{-5},3+2^{-5}]$*, we can conclude that* $(\overline{x}_f,\overline{x}_r) \mathbin{\overline{\prec}} 0$ *is* ***unknown****.*

5.2 AI Abstract Numbers

For estimation of propagations of REs based on AI, we apply AI arithmetic (Definition 24) to Float64-to-Fixed16 ORE arithmetic (Definition 11).

Definition 34 *The set* $\ddot{\Phi}$ *of* ***AI abstract numbers*** *is:*

$$\ddot{\Phi} = \{(fxp, rdf) | fxp, rdf \in \ddot{R}\}$$

AI abstract arithmetic *consists of* $\{\ddot{\boxplus},\ \ddot{\boxminus},\ \ddot{\boxtimes},\ \ddot{\boxdiv}\}$ *on pairs of AI abstract numbers. Let AI abstract numbers* $(\ddot{x}_f,\ \ddot{x}_r),\ \ (\ddot{y}_f,\ \ddot{y}_r) \in \ddot{\Phi}$*. The AI abstract operators are defined below:*

$$\begin{aligned}(\ddot{x}_f,\ \ddot{x}_r) \mathbin{\ddot{\boxplus}} (\ddot{y}_f,\ \ddot{y}_r)&=(\ddot{x}_f \mathbin{\ddot{+}} \ddot{y}_f,\ \ddot{x}_r \mathbin{\ddot{+}} \ddot{y}_r \mathbin{\ddot{+}} \delta_+\varepsilon_p)\\ (\ddot{x}_f,\ \ddot{x}_r) \mathbin{\ddot{\boxminus}} (\ddot{y}_f,\ \ddot{y}_r)&=(\ddot{x}_f \mathbin{\ddot{-}} \ddot{y}_f,\ \ddot{x}_r \mathbin{\ddot{-}} \ddot{y}_r \mathbin{\ddot{+}} \delta_-\varepsilon_p)\\ (\ddot{x}_f,\ddot{x}_r) \mathbin{\ddot{\boxtimes}} (\ddot{y}_f,\ddot{y}_r)\ &=(\ddot{x}_f \mathbin{\ddot{\times}} \ddot{y}_f,\ \ddot{x}_r \mathbin{\ddot{\times}} \ddot{y}_f \mathbin{\ddot{+}} \ddot{x}_f \mathbin{\ddot{\times}} \ddot{y}_r \mathbin{\ddot{+}} \ddot{x}_r \mathbin{\ddot{\times}} \ddot{y}_r \mathbin{\ddot{+}} \delta_\times\varepsilon_p)\\ (\ddot{x}_f,\ \ddot{x}_r) \mathbin{\ddot{\boxdiv}} (\ddot{y}_f,\ \ddot{y}_r)&=(\ddot{x}_f \mathbin{\ddot{\div}} \ddot{y}_f,\ (\ddot{x}_f \mathbin{\ddot{+}} \ddot{x}_r) \mathbin{\ddot{\div}} (\ddot{y}_f \mathbin{\ddot{+}} \ddot{y}_r) \mathbin{\ddot{-}} \ddot{x}_f \mathbin{\ddot{\div}} \ddot{y}_f \mathbin{\ddot{+}} \delta_\div\varepsilon_p)\end{aligned}$$

where$\delta_+, \delta_-, \delta_\times, \delta_\div$ *are given in Equation 2.2 and* ε_p *is a new noise symbol.*

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

Example 26 *The AI abstract numbers* $(\ddot{x}_f, \ddot{x}_r)$*,* $(\ddot{y}_f, \ddot{y}_r)$ *for variables x, y, respectively, in Example 3 are*

$$\begin{cases} \ddot{x}_f = 1 + 2\varepsilon_1 \\ \ddot{x}_r = 2^{-5}\varepsilon_3 \\ \ddot{y}_f = 10\varepsilon_2 \\ \ddot{y}_r = 2^{-5}\varepsilon_4 \end{cases}$$

since $fp = 5$*, the REs of x and y are in* $[-2^{-fp-1}, 2^{-fp-1}] = [-2^{-5}, 2^{-5}]$ $(= [-0.03125, 0.03125])$.

Then, at the line 2 ($rst = x * x$*),* $(\ddot{\mathtt{rst}}_f, \ddot{\mathtt{rst}}_r) = (\ddot{x}_f, \ddot{x}_r) \mathbin{\ddot{\boxtimes}} (\ddot{x}_f, \ddot{x}_r)$ is projected as

follows:

$$\begin{aligned}\ddot{\mathtt{rst}}_f &= \ddot{x}_f \ddot{\times} \ddot{x}_f \\ &= (1 \ + \ 2\varepsilon_1) \ddot{\times} (1 \ + \ 2\varepsilon_1) \\ &= 1 \ + \ 4\varepsilon_1 \ + \ 4\varepsilon_6\end{aligned}$$

$$\begin{aligned}\ddot{\mathtt{rst}}_r &= 2\ddot{\times}\ddot{x}_f\ddot{\times}\ddot{x}_r \ \ddot{+} \ \ddot{x}_r\ddot{\times}\ddot{x}_r\ddot{+}\delta\varepsilon_5 \\ &= 0.031250\varepsilon_5 \ + \ 0.062500\varepsilon_2 \ + \ 0.12500\varepsilon_7 \ + \ 0.000976563\varepsilon_8\end{aligned}$$

where ε_5 is introduced for RE of the fixed point multiplication, ε_6, ε_7, ε_8 are introduce for AI multiplications.

AI Comparison Operators

Replacing (x_f, x_r), (y_f, y_r) in Float-to-fixed ORE comparisons (Definition 8) with AI abstract numbers $(\ddot{x}_f, \ddot{x}_r)$, $(\ddot{y}_f, \ddot{y}_r)$ we obtain AI abstract comparison operators:

Definition 35 *Let $(\ddot{x}_f, \ddot{x}_r)$, $(\ddot{y}_f, \ddot{y}_r) \in \ddot{\Phi}$, let $\overline{x}_f, \overline{x}_r$ be AI projections of $\ddot{x}_f, \ddot{x}_r$, and let $\overline{y}_f, \overline{y}_r$ be AI projections of $\ddot{x}_f, \ddot{x}_r$.*

$$((\ddot{x}_f, \ddot{x}_r) \ddot{<} (\ddot{y}_f, \ddot{y}_r)) = ((\overline{x}_f, \overline{x}_r)\overline{<}(\overline{y}_f, \overline{y}_r))$$

$$((\ddot{x}_f, \ddot{x}_r)\ddot{=}(\ddot{y}_f, \ddot{y}_r)) = ((\overline{x}_f, \overline{x}_r)\overline{=}(\overline{y}_f, \overline{y}_r))$$

The following example illustrates how to evaluate AI abstract comparison $\ddot{<}$.

Example 27 *Use $(\ddot{x}_f, \ddot{x}_r)$, and $(\ddot{y}_f, \ddot{y}_r)$ as in Example 26. $(\ddot{x}_f, \ddot{x}_r) \ddot{<} 0$ is evaluated as follows:*

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

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

5.3 EAI Abstract Numbers

For estimation of propagations of REs based on EAI, we apply EAI arithmetic (Definition 26) to Float64-to-Fixed16 ORE arithmetic (Definition 11).

Definition 36 *The set $\hat{\Phi}$ of **EAI abstract numbers** is:*

$$\widehat{\Phi} = \{(fxp, rdf) | fxp, rdf \in \widehat{R}\}$$

EAI abstract arithmetic *consists of* $\{\widehat{\boxplus}, \widehat{\boxminus}, \widehat{\boxtimes}, \widehat{\div}\}$ *on pairs of EAI abstract numbers. Let EAI abstract numbers* $(\hat{x}_f,\ \hat{x}_r),\ (\hat{y}_f,\ \hat{y}_r) \in \widehat{\Phi}$*. The EAI abstract operators are defined below:*

$$\begin{aligned}
(\hat{x}_f,\ \hat{x}_r) \mathbin{\widehat{\boxplus}} (\hat{y}_f,\ \hat{y}_r) &= (\hat{x}_f \mathbin{\hat{+}} \hat{y}_f,\ \hat{x}_r \mathbin{\hat{+}} \hat{y}_r \mathbin{\hat{+}} \delta_+) \\
(\hat{x}_f,\ \hat{x}_r) \mathbin{\widehat{\boxminus}} (\hat{y}_f,\ \hat{y}_r) &= (\hat{x}_f \mathbin{\hat{-}} \hat{y}_f,\ \hat{x}_r \mathbin{\hat{-}} \hat{y}_r \mathbin{\hat{+}} \delta_-) \\
(\hat{x}_f,\ \hat{x}_r) \mathbin{\widehat{\boxtimes}} (\hat{y}_f,\ \hat{y}_r) &= (\hat{x}_f \mathbin{\hat{\times}} \hat{y}_f,\ \hat{x}_r \mathbin{\hat{\times}} \hat{y}_f \mathbin{\hat{+}} \hat{x}_f \mathbin{\hat{\times}} \hat{y}_r \mathbin{\hat{+}} \hat{x}_r \mathbin{\hat{\times}} \hat{y}_r \mathbin{\hat{+}} \delta_\times) \\
(\hat{x}_f,\ \hat{x}_r) \mathbin{\widehat{\div}} (\hat{y}_f,\ \hat{y}_r) &= (\hat{x}_f \mathbin{\hat{\div}} \hat{y}_f,\ (\hat{x}_f \mathbin{\hat{+}} \hat{x}_r) \mathbin{\hat{\div}} (\hat{y}_f \mathbin{\hat{+}} \hat{y}_r) \mathbin{\hat{-}} \hat{x}_f \mathbin{\hat{\div}} \hat{y}_f \mathbin{\hat{+}} \delta_\div)
\end{aligned}$$

where $\delta_+, \delta_-, \delta_\times, \delta_\div$ *are given in Equation 2.2.*

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

Example 28 *The EAI abstract numbers* $(\hat{x}_f, \hat{x}_r)$, $(\hat{y}_f, \hat{y}_r)$ *for variables x, y, respectively, in Example 3 are*

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

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 ($rst = x * x$*),* $(\widehat{\texttt{rst}_f}, \widehat{\texttt{rst}_r}) = (\hat{x}_f, \hat{x}_r) \mathbin{\widehat{\boxtimes}} (\hat{x}_f, \hat{x}_r)$ is projected as follows:

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

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

EAI abstract comparison operators

Replacing (x_f, x_r), (y_f, y_r) in Float-to-fixed comparison operators (Definition 8) with EAI abstract numbers $(\hat{x}_f, \hat{x}_r)$, $(\hat{y}_f, \hat{y}_r)$, we obtain EAI abstract comparison operators:

Definition 37 *Let* $(\hat{x}_f, \hat{x}_r),\ (\hat{y}_f, \hat{y}_r) \in \widehat{\Phi}$*, let* $\overline{x}_f, \overline{x}_r$ *be EAI projections of* $(\hat{x}_f,\ \hat{x}_r)$*, and let* $\overline{y}_f, \overline{y}_r$ *be EAI projections of* $(\hat{y}_f,\ \hat{y}_r)$*.*

$$((\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{<}} (\widehat{y}_f, \widehat{y}_r)) = ((\overline{x}_f, \overline{x}_r) \mathbin{\overline{<}} (\overline{y}_f, \overline{y}_r))$$

$$((\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{=}} (\widehat{y}_f, \widehat{y}_r)) = ((\overline{x}_f, \overline{x}_r) \equiv (\overline{y}_f, \overline{y}_r))$$

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

Example 29 *Use* $(\widehat{x}_f, \widehat{x}_r)$*, and* $(\widehat{y}_f, \widehat{y}_r)$ *as in Example 28.* $(\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{<}} 0$ *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* $(\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{<}} (\widehat{y}_f, \widehat{y}_r)$ *is* ***unknown****.*

5.4 Meet Operator

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 38 *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: $(u_0 + \sum_{i=1}^{n} u_i \varepsilon_i) \mathbin{\ddot{\cup}} (v_0 + \sum_{i=1}^{n} v_i \varepsilon_i) = (\frac{u_0+v_0}{2} + \frac{|u_0-v_0|}{2} \varepsilon_{n+1} + \sum_{i=1}^{n} t_i \varepsilon_i)$ *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: $(\overline{u}_0 + \sum_{i=1}^{n} \overline{u}_i \varepsilon_i) \mathbin{\widehat{\cup}} (\overline{v}_0 + \sum_{i=1}^{n} \overline{v}_i \varepsilon_i) = (\overline{u}_0 \mathbin{\overline{\cup}} \overline{v}_0) + \sum_{i=1}^{n} (\overline{u}_i \mathbin{\overline{\cup}} \overline{v}_i) \varepsilon_i$.

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

Definition 39 *The meets in abstract values are:*

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

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

5.5 Abstraction for ORE analysis

Abstract domain

For an ORE problem, we abstract a concrete environment as an abstract domain by using range representations.

Definition 40 *Let Var be the set of all variables of the program. An **abstract domain** at a program location is the set of functions $AbsEnv = \{Var \rightarrow \Phi^k\}$, where $k = |Var|$ and $\Phi \in \{\overline{\Phi}, \ddot{\Phi}, \hat{\Phi}\}$.*

*Let $e, e' \in AbsEnv$, **abstract meet operation** is defined below:*

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

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

Abstract environment transformation

For assignment $x = E$, the abstract value at this location will update the value of x by $val(E)$ where $val(E)$ is the result of evaluating E by using CI abstract arithmetic (or AI abstract arithmetic or EAI abstract arithmetic). The abstract semantic is:

$$\lambda e.e \bullet x \mapsto val(E)$$

Conditional judgment at the abstract level

For condition $if\ (C)\ then\ S$, the abstract value will not be change. However, it will be used to evaluate C, called value of C is $val(C)$. If $val(C) = false$ then S will not be visited, otherwise (e.i., $val(C) \in \{true, unknow\}$) we assume that $C = true$, and thus S will be visited.

Chapter 6

ORE Analysis as Weighted Model Checking Problem

As can be seen in Chapter 3, dataflow analysis can be treated as weighted model checking and abstraction. In this chapter, we will exploit how this idea of dataflow analysis can be applied to solve the ORE problems. In particular, using weighted model checking as a back-end framework, we first design: a weighted transition system for the ORE problem in Section 6.1 and 6.2. After that, in Section 6.3, we show how the ORE analysis can be reduced to the weighted model checking problem over the resulting weighted transition system. Finally, we implement our ORE analysis framework in a prototype tool called CANA. Details of the implementation and some experimental results are given in Section 6.4.

6.1 Weighted Domain for the ORE Problem

Weight Design

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

Definition 41 *The weight domain (bounded idempotent semiring)* $\mathcal{S} = (D, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ *with*

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

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

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 that the target programs have bounded loops only; thus after unfolding loops, we obtain acyclic transition systems. Further, note that 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. In particular, we:

- restrict the models to be acyclic,
- fix an initial abstract environment val_0, and

Then, the weights can be generated on-the-fly as in Subsection 3.2.3.

6.2 Weighted Transitions for the ORE Problem

The inputs of our analysis are a subclass of C programs that have bounded loops only. In preprocessing phase, C programs are transformed into three address codes. Next, analysis is performed on these three address codes. Basically, the instructions of three address codes include:

- **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.

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

Table 6.1: Weight function of ORE analysis

- **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. Thus, the generated transition system is *acyclic*.

After preprocessing phase, we obtain an acyclic transition system $\mathcal{P}$ and the initial abstract environment $I \in AbsEnv$. The weight function is defined as follows:

Definition 42 *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 1.*

As a result, we obtain the weighted transition system:

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

The following example describes how to create the weighted transition system for the program in Example 3.

Example 30 *We use EAI representation type.*

The CIL codes of the C program in Fig. 1.2 are shown in Fig. 6.1. 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 $\hat{v}^{(i)}_f$, $\hat{v}^{(i)}_r$ respectively.

The transition system is $\mathcal{P} = (P, \Delta)$, where $P = \{st1, st2..., st6\}$, Δ is shown in Fig. 6.2 and f is defined in Table 6.2.

The initial abstract environment W_{init} at $st1$ is generated from initial range values of variables (given in the topmost comments in Fig. 1.2), in that:

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

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

Figure 6.1: CIL code for Example 3

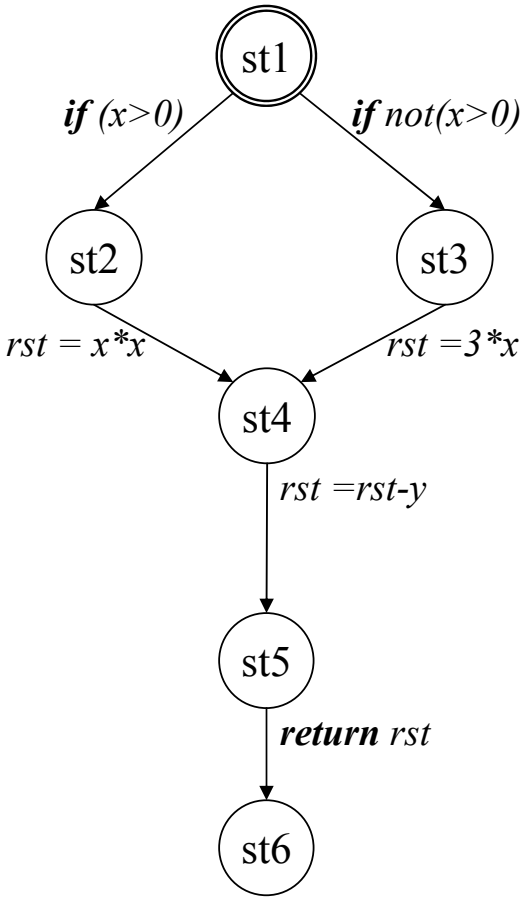


Figure 6.2: CFG of three address codes in Fig. 6.1

transition	weight
(st0,st1)	$(W_{init},\ W_{init})$
(st1,st2)	$if\ ((x \widehat{>} 0) = false)\ \mathbf{0}^+\ else\ \mathbf{1}^+$
(st1,st3)	$if\ not((x \widehat{>} 0) = false)\ \mathbf{0}^+\ else\ \mathbf{1}^+$
(st2,st4)	$(W_i,\ \{rst_o = x_i \mathbin{\widehat{\boxtimes}} x_i,$ $v_o = v_i \mid v \in Var \backslash \{rst\}\})$
(st3,st4)	$(W_i,\ \{rst_o = 3 \mathbin{\widehat{\boxtimes}} x_i,$ $v_o = v_i \mid v \in Var \backslash \{rst\}\})$
(st4,st5)	$(W_i,\ \{rst_o = rst_i \mathbin{\widehat{\boxminus}} y_i,$ $v_o = v_i \mid v \in Var \backslash \{rst\}\})$
(st5,st6)	$\mathbf{1}^+$

Table 6.2: Weight function for a CIL code in Example 30

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

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

Theorem 3 *For a C (CIL) program with bounded loops only, ORE analysis is sound.*

6.3 ORE Analysis

The ORE analysis problem will be solved as weighted model checking on acyclic models by the following steps (Fig. 6.3).

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.

The following example illustrates how to analyze ORE of a C program.

Example 31 *The input ranges of* x *and* y *(in Example 3) are represented by*

$$\widehat{x}_f = [1,\ 1]\ +\ [2,\ 2]\varepsilon_1 \quad and\ \ \widehat{x}_r = [2^{-5},\ 2^{-5}]\varepsilon_3,$$
$$\widehat{y}_f = [0,\ 0]\ +\ [10,\ 10]\varepsilon_2 \quad and\ \ \widehat{y}_r = [2^{-5},\ 2^{-5}]\varepsilon_4.$$

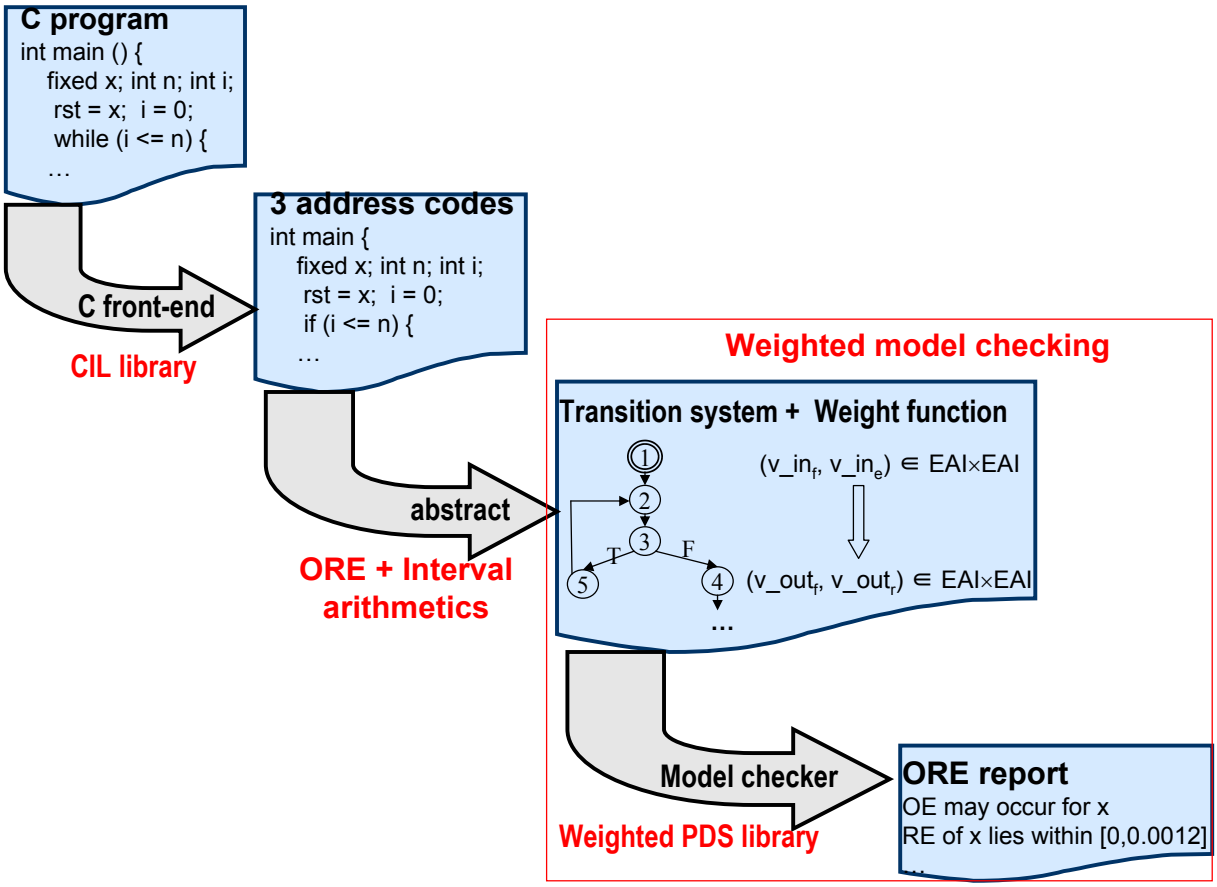


Figure 6.3: ORE analysis as weighted model checking

Since $fp = 4$*, the initial REs of* x *and* y *lie in* $[-2^{-5}, 2^{-5}]$.

At line (1), since the initial range of x *is* $[-1, 3]$*, CANA cannot decide* $(x > 0)$*. Therefore, it traces both line (2) and line (3), and later merges their results before line (4).*

At lines (2) and (3), the REs of rst are computed by $\hat{\boxtimes}$.

$$\widehat{rst}_r^{(2)} = [-0.031250, 0.031250] + [-0.123091, 0.123091]\varepsilon_1 + [0.059615, 0.065385]\varepsilon_3$$
$$\widehat{rst}_r^{(3)} = 3 \mathbin{\hat{\times}} \hat{x}_r = [0.093750, 0.093750]\varepsilon_3$$

They are merged as:

$$\begin{aligned}\widehat{rst}_r^{(2,3)} &= [-0.031250, 0.031250] + [-0.123091, 0.123091]\varepsilon_1 + ([0.059615, 0.065385] \cup \\ &\quad [0.093750, 0.093750])\varepsilon_3 \\ &= [-0.031250, 0.031250] + [-0.123091, 0.123091]\varepsilon_1 + [0.059615, 0.093750]\varepsilon_3\end{aligned}$$

At line (4), we obtain the RE of rst by $\hat{\boxminus}$*:*

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

The RE $\hat{r}$ *of rst (i.e.,* $\widehat{rst}_r^{(5)}$*) coincides with* $\widehat{rst}_r^{(4)}$*, and is bounded by* $[-0.279341, 0.279341]$*. We denote*

$$\hat{r} = \overline{r}_0 \ + \ \sum_{i=1}^{4} \overline{r}_i \varepsilon_i$$

and refer to the coefficient CI of ε_i *(in* $\widehat{rst}_r^{(4)}$*) by* $\overline{r}_i$*.*

Note that, during this analysis, over approximations occur at the conditional branch (line (1)) with $\cup$ *and the multiplication (line (2)) with* $\hat{\delta} = [-0.031250, 0.031250]$*.*

The above example shows that the over approximations will occur at (1) nonlinear operations; (2) undecided conditional branch. In case the conditional branch is decided, analysis will not over approximate it. The following example shows a case in that analysis will not over approximate the conditional branch.

Example 32 *In Example 31, if we reduce the initial range of* x_f *to* $[1, 3]$*, the condition* $x > 0$ *is decided to be true at line (1), and* CANA *ensures that st*3 *will not be executed. Then, at line (4), by substraction* $\hat{\boxminus}$*,*

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

Hence, 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 31.

6.4 Implementation and Experiments

CANA Implementation

We have implemented our analysis framework in a tool *C ANAlyzer* (CANA). CANA uses two libraries as back-end engines: CIL library [1] and WPDS library [2].

- CIL (C Intermediate Language) is a high-level representation along with tools that permit source-to-source transformation of C programs. CIL is used to generate three address codes, information about variables, and CFG of C program.

[1] http://hal.cs.berkeley.edu/cil/

[2] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/

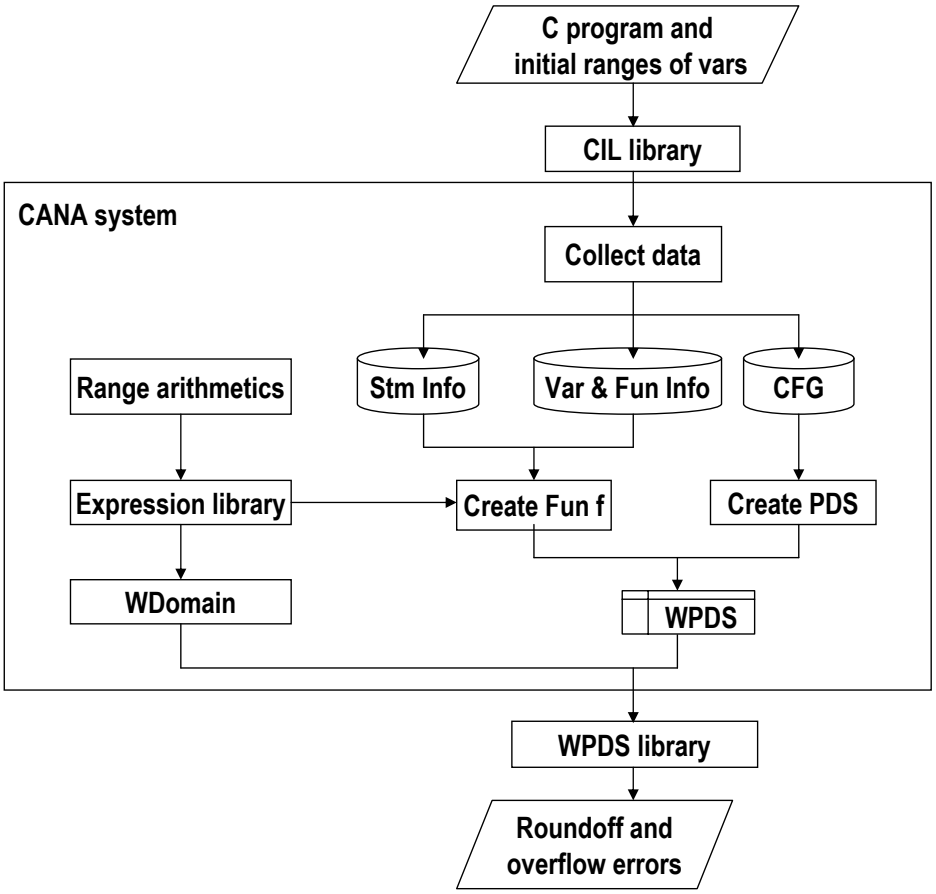


Figure 6.4: CANA system

- 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 at the moment, we adopt WPDS only for weighted finite state (and acyclic) transition systems (i.e., weighted pushdown system with empty stack).

The inputs of CANA are a 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 (Figure 7.4) as follows:

1. *Collect data* module generates information required for 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 range 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 C program.

5. *Create Fun f* module assigns a weight to each transition.

6. *WDomain* module includes two operations: $\otimes$ and $\oplus$.

Preliminary Experiment

We have implemented in CANA three types of range representations: CI, AI, and EAI (on PC with Intel(R) Xeon(TM) CPU 3.60GHz, 3.37Gb of memory). CANA can analyze programs that have nested loops 64×64.

Compare analysis by EAI with analysis by other intervals, general test, and Fixed-point toolbox (Matlab)

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

1. **P2:** program in Figure 1.2
2. **P5:** program that calculates the polynomial of degree 5 $(1-x+3x^2-2x^3+x^4-5x^5)$
3. **Sine:**program that calculates the sine function by Taylor expansion up to degree 21
4. **subMpeg:** a fragment taken from the mpeg4 decoder reference algorithm, consisting of an bounded loop
5. **rump:** program that calculates the Rump function $((333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b})$

We fix the fraction part $fp = 8$ and do experiments for several input ranges. Figure 6.5 shows results of analyzing the program $P2$. The *Method* column shows methods of checking ORE, including:

- General test: execute general test over 8000 test cases (we divide the input domain into equilateral meshes and select one test case from each mesh).

- Matlab test: execute test over 8000 random test cases using Matlab's fixed point toolbox [3]

- CI-CANA: analyze by CANA using CI arithmetic

- AI-CANA: analyze by CANA using AI arithmetic

[3]http://www.mathworks.com/products/fixed/

Method	Input ranges				Time(s)
	([-1,1],[-10,0])	([-1,1],[0,10])	([1,3],[-10,0])	([1,3],[0,10])	
Test	0.00510	0.005201	0.00582	0.00625	0.026
Matlab test	0.01519	0.01528	0.02845	0.02821	1.831
CI-CANA	0.02735	0.02735	0.03126	0.03126	0.129
AI-CANA	0.02735	0.02735	0.03126	0.03126	0.130
EAI-CANA	0.02733	0.02733	0.03126	0.03126	0.130

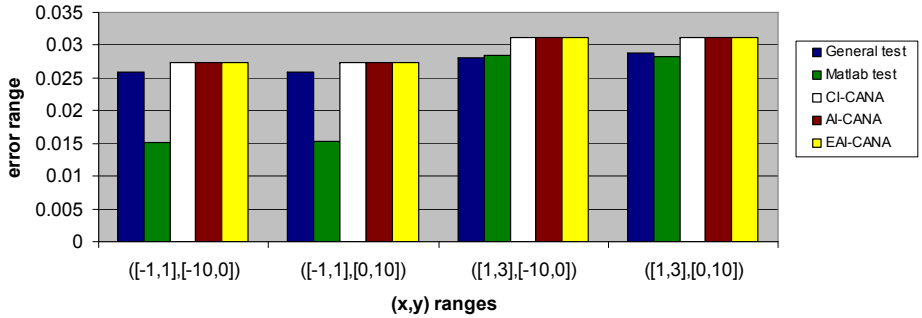


Figure 6.5: The analysis result of P2(x)

- EAI-CANA: analyze by CANA using EAI arithmetic

The *Input ranges* columns are the width of RE ranges which are estimated by each method. The *Time* column shows the average executing time for each case.

Our experiments show that:

- *EAI is more precise than CI and is comparable to AI.* We get similar results for program P5 (Figure 6.6) program Sine (Figure 6.7), program subMpeg (Figure 6.8), and program rump (Figure 6.9).

- The gaps between over approximations (by CANA) and under approximations (by general test and testing based on Fixed-point toolbox, Matlab) are large for programs sin, rump, and are small for others programs.

Note that if we increase the values of input ranges of rump(e.g., $a = 77617$, $b = 33096$), an overflow warning may be display.

Method	Input ranges				Time(s)
	[0,0.2]	[0.2,0.4]	[0.4,0.6]	[0.6,0.8]	
Test	0.01909	0.03000	0.03891	0.06159	0.026
Matlab test	0.01675	0.02527	0.03171	0.05794	1.389
CI-CANA	0.04373	0.05990	0.09172	0.14885	0.129
AI-CANA	0.03770	0.04060	0.06179	0.10447	0.130
EAI-CANA	0.03232	0.03560	0.05763	0.10046	0.130

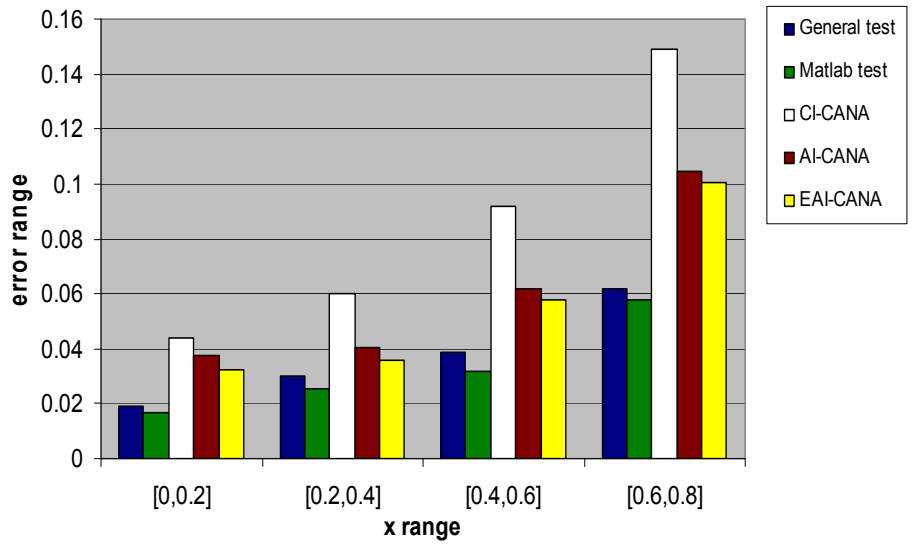


Figure 6.6: The analysis result of P5(x)

Method	Input ranges				Time(s)
	[0,0.2]	[0.2,0.4]	[0.4,0.6]	[0.6,0.8]	
Test	0.00647	0.0108	0.01049	0.01021	0.028
Matlab test	0.00510	0.00763	0.00747	0.00977	1.325
CI-CANA	0.02040	0.02080	0.021390	0.02220	0.140
AI-CANA	0.02035	0.02040	0.02029	0.01998	0.145
EAI-CANA	0.01759	0.01760	0.01749	0.01719	0.144

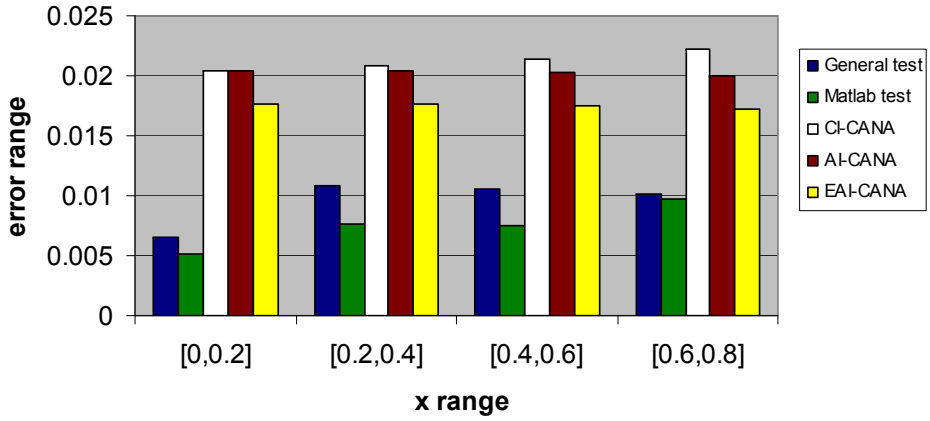


Figure 6.7: The analysis result of Sin(x)

Method	Input ranges				Time(s)
	[0,30]	[30,60]	[60,90]	[90,120]	
Test	1.36309	1.36310	1.36309	1.363086	0.031
Matlab test	1.40570	1.40601	1.40597	1.40575	17.662
CI-CANA	1.73766	1.7377	1.73766	1.73766	0.137
AI-CANA	1.63453	1.6345	1.63453	1.63453	0.142
EAI-CANA	1.63453	1.6345	1.63453	1.63453	0.145

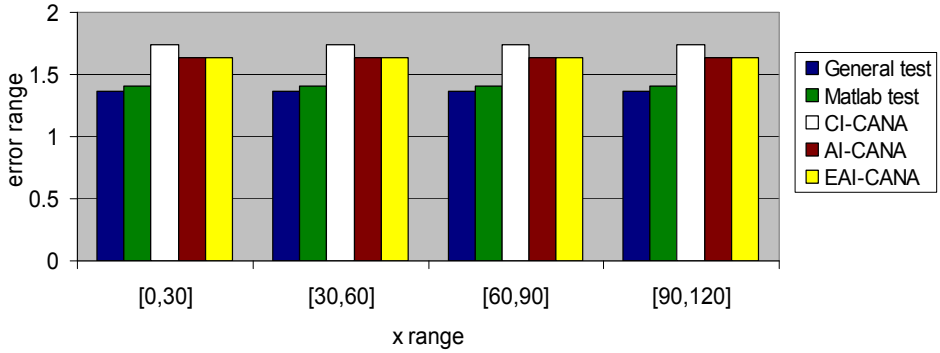


Figure 6.8: The analysis result of subMpeg(exps)

Method	Input ranges				Time(s)
	([0,0.5],[1,1.5])	([0.5,1],[1,1.5])	([0,0.5],[1.5,2])	([0.5,1],[1.5,2])	
Test	58.5123	58.1851	238.364	238.9436	0.023
Matlab test	65.5852	66.3911	293.8237	287.6041	2.169
EAI-CANA	94.6044	100.2262	367.6221	379.861053	0.145

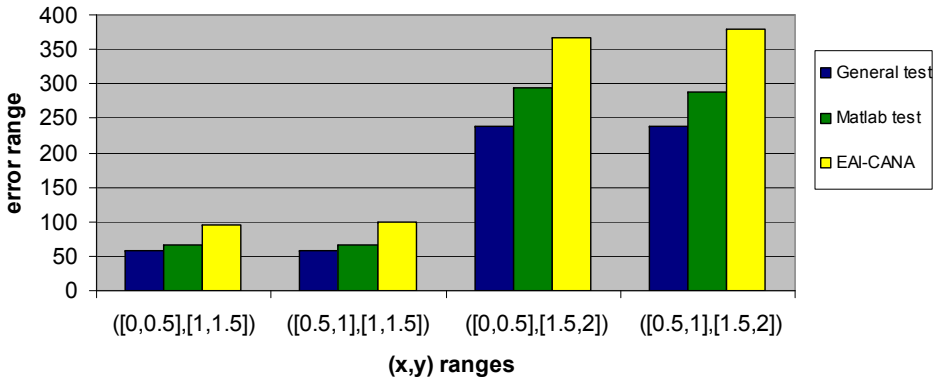


Figure 6.9: The analysis result of rump

Analyzing multiple variables functions by CANA

To consider the effective of CANA, we also do experiment over complex function (i.e., 10 variables, degree $\in [7, 10]$), including:

- **P7:** program that calculates function degree 7 ($10.14a_1a_2a_3 - 11.1a_2a_3 - 0.5a_1a_4 - 0.5a_1^3a_4a_5a_6a_8 + a_6a_7^2 - 5.1a_7a_9 + a_1a_8 + 8.2a_9 - a_{10}$)
- **P8:** program that calculates function degree 8 ($0.3a_1^6a_2a_3 + a_2a_3a_4 - a_2a_4a_5 + a_1^2a_7a_9 + 1.9a_3 - a_1a_6 - a_7 - a_8 - a_9 + a_{10}$)
- **P9:** program that calculates function degree 9 ($0.1a_1^7a_2a_4 + 7.1a_1a_2a_4 - 8.5a_2a_4a_5 + 4.1a_2a_5 - 9.5a_1a_3 + 4a_3 + a_6^2 - 7a_7 + 8a_8 - 9a_9 + 10a_{10}$)
- **P10:** program that calculates function degree 10 ($a_1^2a_2 - 1.1a_1a_2 + 3.1a_2a_3 - 0.1a_1^7a_4a_5a_6 + a_4a_6a_7 - a_1^2a_7 + 0.1a_8 - 0.2(a_9 - a_{10})$)

We compare analysis by EAI with both analysis by standard interval CI as well as under approximation by random testing. Since, the number of variables are large, to get precise testing result, we may need a big number of test cases. For this reason, we perform testing for two cases: 8000 test cases and 80.000.000 test cases.

Figure 6.10 shows the experimental results of the above 4 programs (**P7**, **P8**, **P9**, and **P10** columns) with fixed point format (sign, 5,10), and input ranges $[0, 1]^{10}$. **Method** column shows method of checking OREs, including:

- Random test 1: execute test over 8000 random test cases

Method	P7	P8	P9	P10	Time(s)
Random test 1	0.025851	0.007635	0.038379	0.009711	1
Random test 2	0.037302	0.012215	0.047672	0.014344	150
CI-CANA	0.120768	0.032932	0.112921	0.040071	1
EAI-CANA	0.097232	0.030732	0.096658	0.036410	1

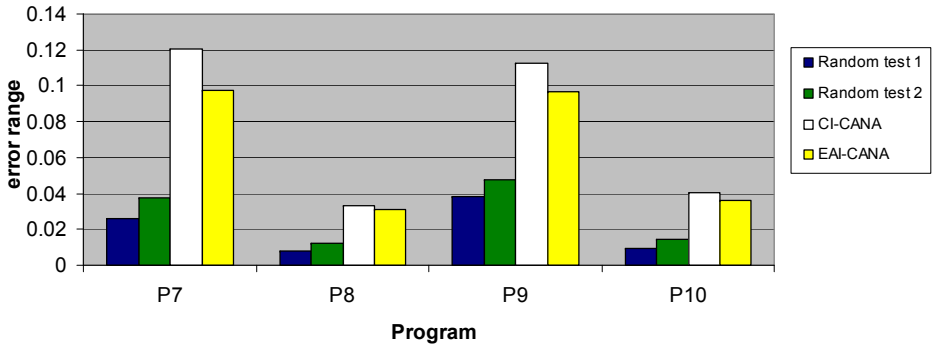


Figure 6.10: The analysis result of 10-variable functions of degree $\geqslant 7$

- Random test 2: execute test over 80.000.000 random test cases
- CI-CANA: analyze by CANA using CI interval
- EAI-CANA: analyze by CANA using EAI interval

The experiments show that EAI is more precise than CI. Moreover, for complex function, the gaps between over approximations (CANA) and under approximations (test) are large.

Chapter 7

Detecting REs based on Counterexample-guided Narrowing

Static analysis is useful in proving safety properties of ORE problem. But it requires over approximation 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) [8]. Inspired by CEGAR, we combine testing and RE analysis.

This chapter 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 [51] which returns an over approximation 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 over approximate 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 chapter, we focus only on roundoff errors. We assume that ORE analyzer (CANA) does not detect any overflow errors. The rest of this chapter is organized

as follows. Section 7.1 presents the overview of the *counterexample-guided narrowing* approach. Section 7.2 introduces techniques to improve test data generation. Section 7.3 propose a technique to improve analysis phase by narrowing input domain based on "divide and conquer" technique. Section 7.4 describes CANAT implementation and reports experiments.

7.1 Counterexample-guided Narrowing Approach

7.1.1 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 $\hat{v}_i = (\frac{l_i+h_i}{2}) + (\frac{h_i-l_i}{2})\varepsilon_i$.

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

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

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 33 *In the analysis result of Example 31:*

- *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.

7.1.2 Counterexample-guided Narrowing Approach

Our approach are broken down into the following steps. First, we apply an overflow and roundoff errors analysis from our previous work [51] which returns an over approximation 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 over approximate 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.

7.2 Refining Test Data Generation

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 43 *For an interval* $[l, h]$ *and* $k \geqslant 1$*,*

- *the* k**-random ticks** *are* $c_1, \cdots, c_k$*, and*
- *the* k**-periodic ticks** *are* $\{c, c + \Delta, \cdots, c + (k-1)\Delta\}$*,*

where $\Delta = \frac{h-l}{k}$*, and* $c \in [l, l + \Delta]$*,* $c_1, \cdots, c_k \in [l, h]$ *are randomly generated.*

Remark 8 *The* k_i*-random ticks and the* k_i*-periodic ticks are used for random testing and counterexample-guided narrowing, respectively. For periodic ticks, the offset* c *is randomly chosen to avoid overlaps in refinement loops.*

Example 34 *Let us consider the C program as Fig. 1.2.*

- *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.

7.2.1 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 44 *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 $\hat{r} = \overline{r}_0 \mp \sum_{i=1}^{2m} \overline{r}_i \varepsilon_i$ 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 4 *If there exists a counterexample in D, then there exists a counterexample in $D_{max} \cup D_{min}$.*

Example 35 *By observation of analysis result $\hat{r}$ in Example 31, we can find D_{max} of the input domain D is*

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

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.

7.2.2 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 45 *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 36 *For* $\hat{r}$ *in Example 31 and* D_{max} *in Example 35, 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,*

$$\begin{aligned} t_1 &= \lceil \tfrac{2 \times 0.123091}{0.01} \rceil = 25 \\ t_2 &= 1 \\ t_3 &= \lceil \tfrac{2 \times 0.09375}{0.01} \rceil = 19 \\ t_4 &= 1 \end{aligned}$$

Thus, the number of test cases is $25 \times 1 \times 19 \times 1 = 475$, *and the RE found by testing* 475 *test cases is* 0.219720.

7.3 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 over approximations:

- 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 44)
- Divide the input ranges (in D_{max} and D_{min}) of the most sensitive noise symbol ε_k into two ranges.

Definition 46 *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 37 *From Example 33, the most sensitive noise symbol is ε_1.*
Form Example 35, 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 33).

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

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

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 satisfies ORE requirement. Note that, before decomposition, it was $[-0.28, 0.28]$ (Example 31), which exceeds the RE threshold bound (Fig. 7.1). 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.

Combining Analysis and Testing Algorithm

Algorithm 7.3 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 over approximate RE in

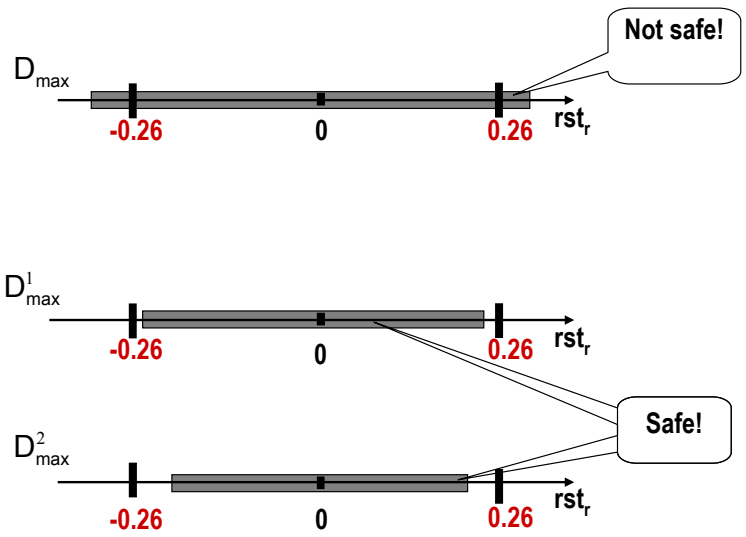


Figure 7.1: Effects of decomposition of D_{max} to D^1_{max}, D^2_{max}

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 $\hat{r}$.

7.4 Implementation and Experiments

CANAT Implementation

A counterexample-guided narrowing framework is implemented as a prototype tool CANAT (*C ANAlyzer and Tester*). An input of CANAT is a C program with nested loops (e.g., 64×64) and arrays with a fixed length (e.g., 64) without procedure calls or pointer manipulations, which typically appears in DSP decoders reference algorithms.

Fig. 7.2 shows the construction of CANAT. CANAT uses three external tools:

- CIL library [1] as a preprocessor,
- Weighted PDS library [2] as a backend weighted model checking engine, and
- CANA [51] as an RE analyzer.

CANAT has 4 main modules as follows:

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

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

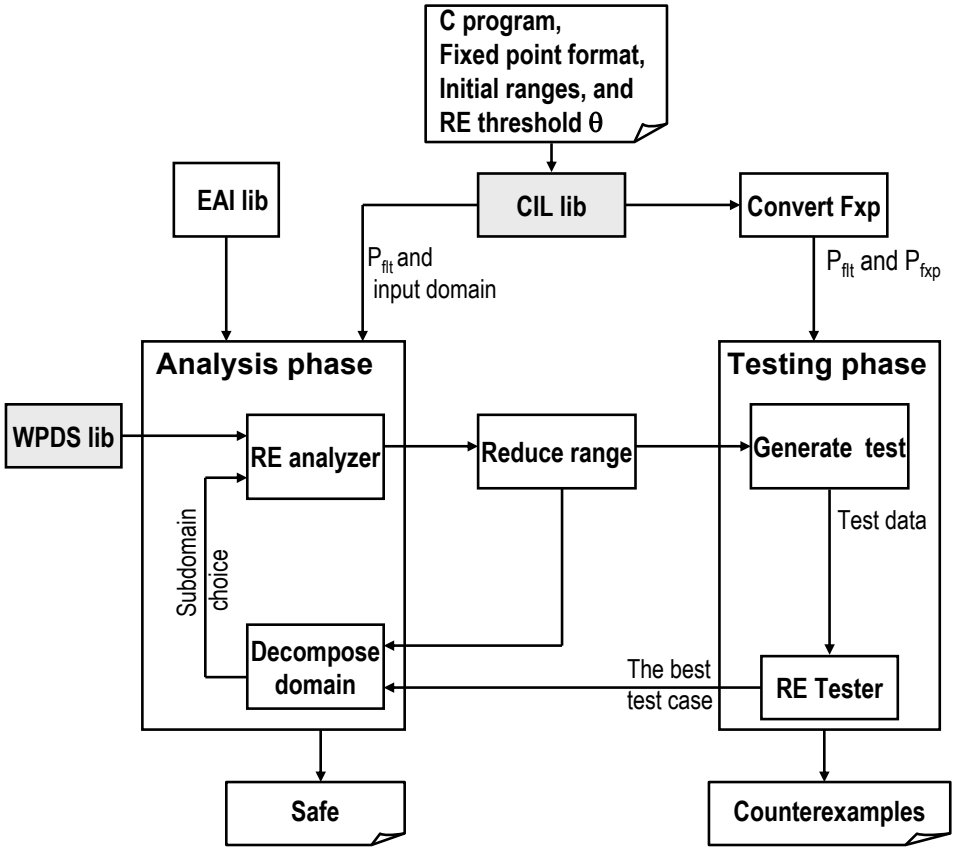


Figure 7.2: CANAT system

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.

Preliminary Experiments

Compare CANAT with single analysis CANA and random test

Table 7.1 shows the results of checking 4 programs (on PC with Intel(R) Xeon(TM)CPU 3.60GHz, 3.37Gb of memory). *The first column* in Table 7.1 shows the names of 4 programs, with the following 40 settings for each program.

Input program	CANA and Random test				CANAT			
	CANA	Rnd test	Time(s)	%Checked	Analysis	Test	Time(s)	%Checked
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%
subMpeg	11	11	65	55.00%	11	19	121	75.00%
rump	11	11	65	55.00%	11	19	121	75.00%

Table 7.1: Compare CANAT with CANA and random test

1. **P2** (Example 3): 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** ($1 - x + 3x^2 - 2x^3 + x^4 - 5x^5$): 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** (by Taylor expansion up to degree 21): 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. **subMpeg** (a fragment taken from the mpeg4 decoder reference algorithm, consisting of an bounded loop): The initial range $[0, 30]$, $fp \in \{7, 8, 9, 10\}$, and $\theta \in \{0.001 + 0.05i \mid 0 \leqslant i \leqslant 9\}$.

Table 7.1 compares experimental results among

- ORE analysis (**CANA**) followed by random testing (**Random test**) with 200 instances, and
- counterexample-guided narrowing by repeating ORE analysis (**Analysis**) and testing (**Test**) 10 times. Each test executes 20 instances.

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.

The second column, **CANA**, and *the sixth column*, **Analysis**, show the number of programs proved to be safe by CANA and CANAT, respectively. *The third column*, **Rnd test**, and *the seventh column*, **Test**, show the numbers of programs in which counterexamples are found by random testing and CANAT, respectively. *The forth and the eighth columns* show the running time, and *the fifth and the ninth columns* show the percents of programs that are shown either to be safe or to have counterexamples.

Although the experiment remains a toy, it shows clear improvement of testing and analysis. Furthermore, the experimental result shows the improvement by counterexample-guided narrowing approach in CANAT, compared to applying an analysis (CANA) or testing alone.

Input program	Matlab			CANAT		
	Checked	Time (s)	%Checked	Checked	Time(s)	%Checked
P2	11	7	65.00%	38	13	95.00%
P5	19	2	47.50%	31	24	77.50%
Sine	5	4	20.00%	29	81	72.50%
subMpeg	23	81	57.70%	30	121	75.00%

Table 7.2: Compare CANAT with Matlab test

Compare CANAT with Fixed-point Toolbox in Matlab [3]

Table 7.2 shows the comparison between CANAT and testing by using Fixed-point toolbox in Matlab, in that the number of test case is 8000.

For the input programs and the setting as in experimental Table 7.1. *The second column* and *the fifth column* show the number of programs can be checked by Matlab testing and CANAT, respectively. *The third column* and *the sixth column* show the running time, and *the fourth and the seventh columns* show the percents of programs that are shown either to be safe or to have counterexamples.

The experimental result shows that CANAT can check more program than testing by Matlab. The reason is CANAT returns both under approximation result (by testing) and over approximation (by analysis with EAI) while testing by Matlab only return under approximation.

Checking multiple variables functions by CANAT

To consider how CANAT can reduce the gap between over approximation (CANA) and under approximation (testing), we check the complex functions from Figure 6.10. In Table 7.3, *the second column*, **CANA**, and *the fifth column*, **Analysis**, show the ranges of RE founds by CANA and CANAT, respectively. *The third column*, **Rnd test**, and *the sixth column*, **Test**, show the ranges of REs found by random testing and CANAT, respectively. *The forth and the seventh columns* show the gap between analysis and testing by CANA-random-test and CANAT, respectively, and *eighth column*, **%Reduced** shows the ratio of the gap between CANAT analysis and CANAT testing to the gap between CANA analysis and random testing.

The experimental result shows that CANAT reduces the gap between over approximation and under approximation.

[3] http://www.mathworks.com/products/fixed/

Input program	**CANA and Random test**			**CANAT**			%Reduced
	CANA	Rnd test	Gap	Analysis	Test	Gap	
P7	0.097232	0.019011	0.078221	0.092208	0.023741	0.068467	87.53
P8	0.030732	0.006724	0.024008	0.028368	0.010272	0.018096	75.37
P9	0.096658	0.014919	0.081739	0.091188	0.029174	0.062014	75.87
P10	0.03641	0.007888	0.028522	0.034258	0.012113	0.022145	77.64

Table 7.3: Checking result of 10-variable functions of degree $\geqslant 7$

Chapter 8

Related Work

The ORE analysis has been attracted extensive attention of research in the numerical analysis recently. Several different approaches has been proposed to deal with the problem of roundoff and overflow problems. In this chapter, we give a survey of recent related works on the topics of ORE analysis.

The ORE problems are one of the central issues in the numerical analysis [17, 22]. There are lots of works on mathematical reasoning to estimate OREs [21, 22], and there is a well-known methodology for the precise addition of floating numbers, which cancels the effect of REs [33]. It is extended to the precise multiplication [53], 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 thesis.

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, 47].
- Octagon, which is surrounded by either vertical, horizontal, or diagonal lines [46].
- Polyhedra, which is represented as the conjunction of linear inequalities [10]. Recently, its refinement SubPolyhedra was proposed [36] 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 [23].

By introducing noise symbols, which preserve dependency of uncertainty, Affine interval (AI) has been proposed [61, 62]. Later, we will see how AI is applied as over approximation for ORE analyses.

Extended Affine interval (EAI) has proposed for under approximation, based on the mean value theorem and Kaucher arithmetic [27]. EAI replaces real coefficients of AI with CI coefficients. We apply this idea for over approximation of ORE analysis.

Numerical Constraint Solvers

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

- iSAT [16], which evaluates non-linear operations to interval constraints by over approximation.
- minismt [64], 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 [30], which applied a variation of polyhedra, called the *bounded vertex representation* for under approximation.

Ganai and Ivancic [18] 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 [25, 20, 37, 48].

The library Apron of numerical abstract domains is also freely available [29].

ORE analysis

For a static analysis, we need a concrete semantics. We obey the semantics of propagation of OREs to [39].

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 [26, 40] (which introduced widening operators to handle loops), except that the octagon abstract domain is used in [45].

They are implemented and showed experimental results. FLUCTUAT is presented in [26] and [45] showed a case study on an embedded avionics software. The technique of [40] is further applied on TMS320 C3X assembler programs [41].

These ORE analyses are over approximation, and easily cause false positives. An ORE analysis with under approximation is proposed based on the mean value theorem and Kaucher arithmetic [27], 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, 31, 32, 44, 58, 59], there are strong demand to solve ORE problems.

For the floating point numbers to fixed point numbers conversion, Fang, et.al. [13, 14, 15] proposed an ORE analysis based on AI, intended for DSP applications. 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. [63] 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 spurious counterexamples in an early stage.

Refining analyses and testing

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

Counterexample Guided Abstraction-refinement (CEGAR) [8] 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.

Chapter 9

Conclusions

Motivated by automatically detecting OREs of hardware systems, this thesis proposes a novel interval-valued approach for representing and reasoning about OREs. We concentrate on the overflow and roundoff errors between floating point arithmetic and fixed point arithmetic, which are frequently troublesome in DSP decoders when a reference algorithm with floating point numbers is converted to a hard-wired logic with fixed point numbers. It contributes several techniques for improving interval computations, automatically estimating OREs with high precision. This chapter reviews the main contributions, and then discusses about future research.

9.1 Summary of the Thesis

First of all, our contributions are summarized as follows:

- In Chapter 4, two new range representations were propose, named extended affine interval (EAI) and positive-noise affine interval (PAI). EAI is extended from AI by assigning for each noise symbol a CI coefficient. By this representation, EAI has two main advantages over current range representations (CI and AI). First, EAI is more precise than CI because EAI can store information about sources of uncertainty as noise symbol, whereas CI cannot. Second, EAI forms are more compact than AI forms. This is because EAI arithmetic does not introduce new noise symbols, while AI arithmetic does. PAI is another way to extend AI in that the noise symbols are set to lie in $[0, 1]$ (instead of $[-1, 1]$). Thus, PAI nonlinear operations are designed to reduce over approximation (compare to AI and EAI).

- In Chapter 6, the ORE analysis is treat as a weighted model checking problem. In that, range representations (i.e., CI, AI, and EAI) are used in the abstraction of OREs. To avoid widening (which often leads to large over approximation), we restrict to a subclass C program which have only bounded loops. The model of the program becomes acyclic model by unfolding loops. Thus, the weigh domain can

be generated by an on-the-fly manner and satisfies the descending chain condition. Based on this approach, a static analysis tool CANA for overflow and roundoff error analysis of subclass C programs was implemented. Although our experiments were performed on small examples, the result is encouraging that EAI is more precise than CI and is comparable to AI.

- In Chapter 7, we proposed an approach for finding both under-approximation and over approximation of RE ranges. In particular, the under-approximation is computed by testing method and over approximation is computed by analysis method. Further, static analysis and testing are combined, to make them refine each other (called *counterexample-guided narrowing*). The refinement loop (i.e., testing + analysis) is repeated to refined both testing phase and analysis phase. Thus, the result will be more precise than either testing or static analysis alone. Note that, this approach can not be applied for detecting REs between real number and floating point numbers (or fixed point numbers) because there is no way to automatically test real algorithm exactly. Using the proposed approach, a prototype tool named CANAT, is implemented based on CANA. We would like to emphasize that, although our experiments are still small, the results are encouraging and show potential usefulness in practice.

9.2 Future Work

For future work, we plan to consider the following issues: For future works, one obstacle is the scalability of CANA and CANAT. We are optimistic since DSP algorithms (e.g., digital video compression [54]) 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 [26] 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 [43], 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.

Bibliography

[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 (2008)

[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] Brown, C. W., Davenport, J. H.: The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proc. of the 2007 International Symposium on Symbolic and algebraic computation*, pp. 54 - 60, ACM (2007)

[7] Clarke, E. Grumberg, O., and Peled, D. A.: Model checking. MIT Press, L.D. (1999)

[8] 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)

[9] 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)

[10] 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)

[11] Dijkstra, E.W.: A discipline of programming. Series in Automatic Computation. Prentice Hall (1976)

[12] Emerson, E. A.: Temporal and modal logic. Handbook of Theoretical Computer Science, MIT (1990)

[13] 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)

[14] 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)

[15] 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)

[16] 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)

[17] Friedman, M. and Kandel, A.: Fundementals of Computer numerical analysis. CRG Press (1994)

[18] 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)

[19] Ganai, M. and Gupta, A.: Completeness in SMT-based BMC for Software Programs. In *Proc. of the International Conference of Design, Automation and Test in Europe*, pp. 831-836, ACM (2008)

[20] 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)

[21] 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)

[22] Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* , pp.5-48, ACM (1991)

[23] 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)

[24] Goodwin,D. W.: Interprocedural dataflow analysis in an executable optimizer. In *Proc. of the Conference on Programming Language Design and Implementation*, pp. 122 - 133, ACM (1997)

[25] Gopan, D.: Numeric program analysis techniques with applications to array analysis and library summarization. Ph. D Thesis, University of Wisconsin-Madison (2007)

[26] 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)

[27] 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)

[28] 754-2008 IEEE Standard for Floating-Point Arithmetic at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4610935. Accessed February 2010.

[29] 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)

[30] 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)

[31] 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)

[32] 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)

[33] Knuth, D.E.: The Art of Computer Programming, Vol. 2: Seminumerical Algorithms. AddisonWesley (2002)

[34] Lacey, D.: Program Transformation using Temporal Logic Specifications. PhD thesis, Oxford University Computing Laboratory (2003)

[35] 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)

[36] 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)

[37] 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)

[38] Loh, E. and Walster, G.W.: Rump's example revisited. Realiable Computing, vol.8 (3), pp.245-248 (2002)

[39] 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)

[40] 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)

[41] 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)

[42] 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)

[43] 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)

[44] 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)

[45] 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)

[46] Mine, A.: The octagon abstract domain. Higher-Order and Symbolic Computation, Vol.19 (1), pp.31-100, 2006.

[47] Moore, R.E.: Interval Analysis. Prentice-Hall (1966)

[48] 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)

[49] 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)

[50] Nielson, F., Nielson, H. R., Hankin, C.: *Principles of program analysis*. Springer (1998)

[51] 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)

[52] 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)

[53] Ogita, T., Rump, S.M., and Oishi, S.: Accurate sum and dot product. SIAM Sci. Comp, vol.26 (6), pp.1955-1988 (2005)

[54] Peter, S.: Digital Video Compression. mcgraw-Hill (2004)

[55] 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)

[56] 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)

[57] 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)

[58] 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)

[59] Shi, C.: Floating-point to Fixed-point Conversion. Ph. D. Thesis, University of California - Berkeley (2004)

[60] 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)

[61] Stolfi, J.: Self-Validated Numerical Methods and Applications. Ph. D. Dissertation, Computer Science Department, Stanford University (1997)

[62] 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)

[63] 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)

[64] 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).

Publications

[1] Ngoc, D. T. B. and Ogawa, M.: Overflow and Roundoff 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)

[2] Ngoc, D. T. B. and Ogawa, M.: Checking Roundoff Errors based on Counterexample-Guided Narrowing. The *25th IEEE/ACM International Conference on Automated Software Engineering*(to appear as a short paper)

[3] Ngoc, D. T. B. and Ogawa, M.: Checking Overflow and Roundoff Errors via Dataflow Analysis and Testing. *Selectively invited from SEFM 2009 in the special issue of International Journal of System Modeling and Software. Springer Verlag 2010* (submitted)